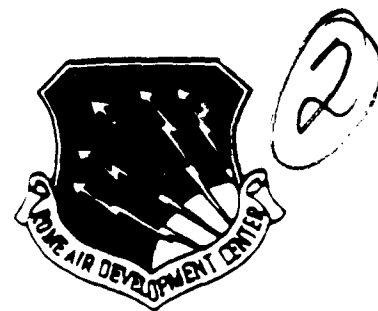


AD-A232 289

RADC-TR-90-101, Vol I (of three)
Final Technical Report
June 1990



DECENTRALIZED COMPUTING TECHNOLOGY FOR FAULT-TOLERANT, SURVIVABLE C3I SYSTEMS

Carnegie-Mellon University

**Sponsored by
Strategic Defense Initiative Office**



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.

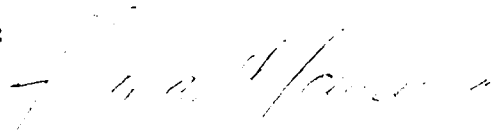
**Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700**

91 3 01 011

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-101, Vol I (of three) has been reviewed and is approved for publication.

APPROVED:



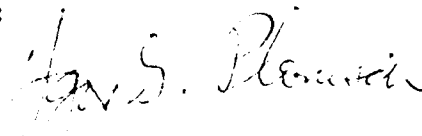
THOMAS F. LAWRENCE
Project Engineer

APPROVED:



Raymond P. Urtz, Jr.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER:



IGOR G. PLONISCH
Directorate of Plans and Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

DECENTRALIZED COMPUTING TECHNOLOGY
FOR FAULT-TOLERANT, SURVIVABLE C³I SYSTEMS

J. Duane Northcutt
Edward J. Burke
James G. Hanko
David P. Maynard
Samuel E. Shipman
Jeffrey E. Trull

E. Douglas Jensen
Raymond K. Clark
Donald C. Lindsay
Franklin D. Reynolds
Jack A. Test

Contractor: Carnegie-Mellon University
Contract Number: F30602-85-C-0274
Effective Date of Contract: 29 Aug 85
Contract Expiration Date: 30 Dec 88
Short Title of Work: Decentralized Computing Technology for Fault-Tolerant, Survivable C³I Systems
Period of Work Covered: Aug 85 - Dec 88
Principal Investigator: E. Douglas Jensen
Phone: (508) 393-2989
Project Engineer: Thomas F. Lawrence
Phone: (315) 330-2158

Approved for public release; distribution unlimited.

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Thomas F. Lawrence (COTD), Griffiss AFB NY 13441-5700, under contract F30602-85-C-0274.

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503.

1 AGENCY USE ONLY (Leave Blank)		2 REPORT DATE June 1990		3 REPORT TYPE AND DATES COVERED Final Aug 85 - Dec 88	
4 TITLE AND SUBTITLE DECENTRALIZED COMPUTING TECHNOLOGY FOR FAULT-TOLERANT, SURVIVABLE C ³ I SYSTEMS				5 FUNDING NUMBERS C - F30602-85-C-0274 PE - 63223C PR - 2300 TA - 02 WU - 10	
6 AUTHOR(S) J. Duane Northcutt, E. Douglas Jensen, Edward J. Burke, Raymond K. Clark, James G. Hanco, Donald C. Lindsay, David P. Maynard, (Cont'd)					
7 PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Carnegie-Mellon University Pittsburgh PA 15213-3890				8 PERFORMING ORGANIZATION REPORT NUMBER	
9 SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Office, Office of the Secretary of Defense Wash DC 20301-7100				10 SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-101, Vol I (of three)	
11 SUPPLEMENTARY NOTES RADC Project Engineer: Thomas F. Lawrence/COTD/(315) 330-2158					
12a DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b DISTRIBUTION CODE	
13 ABSTRACT (Maximum 200 words) Alpha is an operating system for the mission-critical integration and operation of large, complex, distributed, real-time systems. Such systems are becoming increasingly common in both military (e.g., BM/C ³ , combat platform management) and industrial factory and plant automation (e.g., automobile manufacturing) contexts. They differ substantially from the better-known timesharing systems, numerically-oriented supercomputers, and networks of personal workstations. More surprisingly, they also depart significantly from traditional real-time systems, which are predominately for low-level periodic sampled data monitoring and control.					
14 SUBJECT TERMS Real-Time System Distributed Operating System Distributed Computing				15 NUMBER OF PAGES 216	
Decentralized Control Fault-Tolerance Time-Value Functions				16 PRICE CODE	
17 SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18 SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19 SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20 LIMITATION OF ABSTRACT SAR		

6 (Cont'd). Franklin D. Reynolds, Samuel E. Shipman, Jack A. Test, Jeffrey E. Trull

Contents

Preface

Part A: Alpha Overview

Part B: Alpha Requirements and Rationale

Part C: An Example Real-Time Command, Control and Battle Management Application for Alpha



Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Preface

This is the final technical report for contract number F30602-85-C-0274 from the USAF Rome Air Development Center, COTD, covering the period from 1 September 1985 to 31 December 1988. This contract is part of the support for the Archons Project, which is performing research on concepts and techniques for decentralized computing systems. The contract Principal Investigator during the period from 1 September 1985 to 1 October 1987 was E. Douglas Jensen, who is now leading the Archons project from the Concurrent Computer Corporation. This contract's Principal Investigator and Program Manager for the period from 1 October 1987 to 31 December 1988 was J. Duane Northcutt.

This report provides an overview of the Alpha real-time decentralized operating system technology that is being created as part of the Archons Project research. Alpha is the first systems result of the Archons Project, and is a unique operating system focused specifically on real-time command and control applications—e.g., combat platform and battle management.

This report is divided into three parts:

- Alpha Overview
- Alpha Requirements and Rationale
- An Example Real-Time Command, Control and Battle Management Application for Alpha

Table of Contents

1	Introduction	A-2
2	Key Technical Requirements and Approaches	A-3
2.1	Technical Requirements Summary	A-3
2.2	Real-Time	A-4
2.2.1	Time Constraints	A-4
2.2.2	Exceptions	A-7
2.2.3	Determinism	A-8
2.2.4	Guarantees	A-8
2.2.5	Summary of Real-Time in Alpha	A-8
2.3	Distribution	A-9
2.3.1	Distributed Operating Systems	A-9
2.3.2	Distributed Applications	A-10
2.3.3	Distributed Resource Management	A-10
2.4	Survivability	A-11
2.5	Adaptability	A-12
2.6	Programming Model	A-13
2.6.1	Objects	A-13
2.6.2	Operation Invocation	A-13
2.6.3	Threads	A-14
2.6.4	Real-Time Transaction Mechanisms	A-14
2.6.5	Comparison With Other Models	A-16
3	System Structure	A-18
4	Status and Future Plans	A-20
5	Acknowledgments	A-22
6	References	A-23

List of Figures

Figure 1	A Time-Value Function.....	A-5
Figure 2	Variable Time-Value Function.....	A-6
Figure 3	Set of Time-Value Functions.....	A-6
Figure 4	Conventional Distributed Operating System.....	A-9
Figure 5	Distributed Application on a Conventional DOS.....	A-10
Figure 6	“Decentralized” Operating System.....	A-11
Figure 7	Example Alpha Object.....	A-14
Figure 8	An Alpha Thread and its Attributes.....	A-15
Figure 9	Example of Alpha’s Object/Thread Programming Model	A-16
Figure 10	Alpha Releases 2 and 3 Structure	A-18
Figure 11	Alpha OS Prototype (Release 1) Testbed.....	A-19
Figure 12	Alpha OS Release 2+ Initial Testbed	A-20

Abstract

Alpha is an operating system for the mission-critical integration and operation of large, complex, distributed, real-time systems. Such systems are becoming increasingly common in both military (e.g., BM/C³, combat platform management) and industrial factory and plant automation (e.g., automobile manufacturing) contexts. They differ substantially from the better-known timesharing systems, numerically-oriented supercomputers, and networks of personal workstations. More surprisingly, they also depart significantly from traditional real-time systems, which are predominately for low-level periodic sampled data monitoring and control.

The most challenging technical requirements dictated by this application domain are in the areas of: satisfying critical real-time constraints despite the system's inherently stochastic and nondeterministic nature; system-wide (inter-node) run-time resource management in support of distributed programming; survivability despite failures and even attacks; and adaptability to a wide range of ever-changing requirements over decades of use. Previous attempts to address this problem space have produced custom, special-purpose operating systems; for Alpha to satisfy these requirements as a standard, multi-purpose product entails the creation of much unorthodox technology. This includes: using actual application-specified task completion time constraints and functional criticalities directly to manage all resources; employing "best effort" scheduling algorithms which accommodate hard and soft time constraints, dynamic dependencies, and overloads; optimizing performance for the high stress, mission-critical exception cases (such as emergencies), rather than for the most frequent, routine, cases; supporting consistency of distributed data and correctness of distributed actions, within both the OS and the application, through real-time transaction mechanisms in the kernel which independently provide atomicity, application-specific concurrency control, and permanence according to higher level policies; and an object-oriented distributed programming model based on threads which cross object and (transparently and reliably) node boundaries, carrying attributes such as urgency, importance, and reliability, to facilitate global resource management.

Alpha embodies results from nine years of research performed by the Archons Project at Carnegie Mellon University's Computer Science Department, where a prototype was built from 1985 to 1987. General Dynamics/Ft. Worth has successfully written and demonstrated real-time application software on another copy of the system there. Alpha research is ongoing at CMU, and is starting at MIT and several industrial institutions. Now the Alpha effort is led by Concurrent Computer Corporation, where it continues to be sponsored in part by DoD. A series of increasing functionality, next-generation designs and implementations on MIPS-based multiprocessor nodes interconnected with FDDI will be delivered to various Government and industry labs for experimental use beginning in 1990. Concurrent and its strategic partners also plan to provide additional versions of Alpha, including a multilevel secure version, a POSIX-compliant version, and a subset version optimized for conventional low-level sampled data subsystems. Alpha is non-proprietary and portable, and will be the basis for future commercial OS products from Concurrent and other corporations.

1 Introduction

The Alpha operating system project [Jensen 88c] is an uncommon blend of both research and development. Alpha initially arose from Jensen's Archons Project to create new paradigms for real-time distributed computer systems at Carnegie-Mellon University's Computer Science Department [Jensen 79, Jensen 84]. It formed the systems context for designing, implementing, and empirically evaluating unconventional ideas in real-time distributed operating systems. The Alpha prototype was started in 1984, and has been fully operational there since 1987 [Northcutt 87]. It runs directly on the bare hardware of multiprocessor nodes built from modified Sun Microsystems boards and interconnected with Ethernet. It is also installed at General Dynamics/Ft. Worth; it was the basis for their proposed mission management OS onboard the Advanced Tactical Fighter, and in addition they have successfully written and demonstrated real-time distributed battle management/C² application software on it [Clark 88, Northcutt 88c].

The focus of Alpha activity moved with its Principal Investigator to Concurrent Computer Corporation's Westford (Boston) MA facility when they merged with MASSCOMP in the Fall of 1988. It continues to be funded in part there by DoD. Concurrent's own industrial research efforts are also complemented by academic research on Alpha-related topics at CMU and (beginning in Fall 1989) MIT, supported both by DoD and Concurrent. Alpha is non-proprietary and in the public domain for U.S. Government use; it is portable, and source as well as binary licenses will be available for commercial purposes. Alpha's current status and future directions are summarized at the end of this report.

2 Key Technical Requirements and Approaches

2.1 Technical Requirements Summary

The Alpha OS is specifically optimized for performing the mission-critical integration and operation of large, complex, distributed real-time systems [Jensen 88b]. This encompasses: real-time supervisory control—for example, mission (and sometimes vehicle) management of combat platforms in subsurface (e.g., Seawolf), surface (e.g., Aegis, [Jensen 76b]), air (e.g., ATF), and space (e.g., BSTS) environments, together with battle management (e.g., SDI's SDS); plus real-time decision support and information management—for example, C³. Typical application functions in the DoD context include sensor management, multisensor data correlation and fusion, track initiation and maintenance, object identification, situation awareness, threat evaluation, engagement planning, force coordination, stores management, weapons assignment, weapons guidance, and kill assessment.

The characteristics of this application domain differ substantively in almost every respect from those of the more widely known non-real-time systems, such as networked personal productivity workstations and throughput-oriented numerically-intensive computers. What may be surprising is that these characteristics also depart dramatically from those of the traditional real-time context—relatively small, simple subsystems for low-level sampled data monitoring and control (e.g., signal processing, avionics flight control). The mission-critical integration and operation of large, complex, distributed real-time systems (which we will henceforth abbreviate as simply “SIO”) is a completely different, and much more difficult, kind of OS problem compared with conventional real-time systems.

Many of these differences and difficulties stem from the fact that SIO is supervisory level—the data sources and sinks are primarily not sensors and actuators but instead low-level sampled-data subsystems, man-machine interface subsystems, and interconnections to other systems [Boebert 78]. System integration and operation applications involve the OS carrying out activities such as: global resource management; control and coordination; fault, error, and failure recovery; interoperability with lower level and higher level systems; performance evaluation; and man/machine interface. Thus, the application and OS tasks are predominantly aperiodic and asynchronous rather than simply cyclic, and still they have time constraints which are critical to the degree of mission success. The system's large size, distribution, and complexity also contribute to the challenging problems—its behavior is intrinsically dynamic and nondeterministic (although this may not necessarily be directly visible as such to the application). Consequently, stochastic demand for resources frequently exceeds the supply, so conflicts (e.g., dependencies, precedence constraints) inevitably occur, and must be resolved, at execution time.

These attributes of the SIO domain are manifest primarily in four areas of operating system technical requirements [Northcutt 88a]:

- Real-Time: meeting as many as possible of the most important aperiodic as well as periodic time constraints, despite dynamic and stochastic runtime resource contention, overloads, and faults
- Distribution: focusing multiple physically dispersed computing nodes on the exe-

cution of large, complex, distributed computations to perform a mission

- Survivability: preserving the mission, human life, and property in a hostile environment with limited or no repairs or scheduled downtime during missions up to decades long
- Adaptability: serving a wide variety of applications, each of whose requirements evolve continuously over a lifetime of decades, on a dynamic technology base.

Each of these four areas is addressed in the following Sections.

2.2 Real-Time

The term “real-time” is most frequently associated with low-level sampled-data monitoring and control of periodic physical processes—i.e., dataflow/pipelined signal processing, and sensor/actuator feedback control. The vast majority of these applications are relatively small, simple subsystems (often single-algorithm). Their most distinguishing characteristic is universally considered to be rigidly deterministic behavior.

The approach taken in Alpha to dealing with real-time in large, complex, distributed systems is dramatically unorthodox with respect to some of the keystone aspects of traditional real-time—time constraints, exceptions, determinism, and guarantees.

2.2.1 Time Constraints

In SIO, most aperiodic as well as periodic tasks have time constraints composed of two orthogonal components—urgency (i.e., time criticality) and importance (i.e., functional criticality). The historical imposition by OS designers of single-dimensional priorities for the perceived sake of computational simplicity has lost (or at best, confused) that valuable distinction. This has been tolerable in smaller, simpler applications, but as system size and complexity increase, it accounts for a major portion of disproportionately increasing design, integration, test, and modification costs.

Contrary to popular belief, simply starting tasks as fast as possible (which low latency interrupt handling and context switching are meant to ensure) is not the OS performance metric of interest to users. Rapid interrupt response and context switches are necessary means, but not sufficient. The useful metric is *completing* tasks at the most valuable times, neither too early nor too late. In the smaller, simpler applications which dominate real-time, it is usually more or less feasible to arrange that starting a task quickly implies its timely completion, since static resource management techniques (e.g., [Stankovic 87]) assure that there are few, if any, runtime impediments.

For the larger, more complex applications typical of SIO, the requirement is further that as many as possible of the most important tasks (whether aperiodic or periodic) must complete as close as possible to their most valuable times; and this despite dynamic resource demands and conflicts, overloads, and faults. The application software does not (and should not) have access to the kinds of system resources necessary to accomplish all that itself, so the OS must accept responsibility for not just starting each activity, but also for shepherding it through to completion within a useful timeframe. The customary notion of priorities is again inadequate for SIO because it is not expressive enough to convey the meaning of “completion at the most valuable time.” Under these difficult but realistic conditions, an assurance that the OS will always make a “best effort” to

achieve maximal utility, according to application-defined criteria, from the presently available assets, is both the strongest one possible as well as a far stronger one than offered by conventional real-time OS's.

Although SIO includes many mission-critical hard real-time tasks, the majority would be placed in the "soft real-time" category according to much of the literature (although there is considerable confusion there on these terms—e.g., between hard real-time *tasks* and hard real-time *systems*). Most academic real-time publications consider "hard real-time" systems to be those in which every task has a deadline that can never be missed [Stankovic 88]. Such systems are notable because they permit rigidly stylized, but analytically tractable, forms of resource scheduling (e.g., rate-monotonic [Liu 73]). However, almost all real applications not only tolerate some tasks missing some deadlines, but in fact also desire the robustness of being able to accommodate transient delays and overloads—many employ schemes for slipping cycles, adjusting deadlines, apportioning tardiness, re-ordering tasks within rate groups, etc.

In Alpha, the above two problems of priorities are overcome by providing for the management of resources directly with application-specified actual urgency and importance constraints. These time constraints are expressed by "time-value functions" for each activity [Stewart 77*, Jensen 85, Locke 86], which express the value to the system of completing a task at any time, as shown in Figure 1. Time-value functions include classical

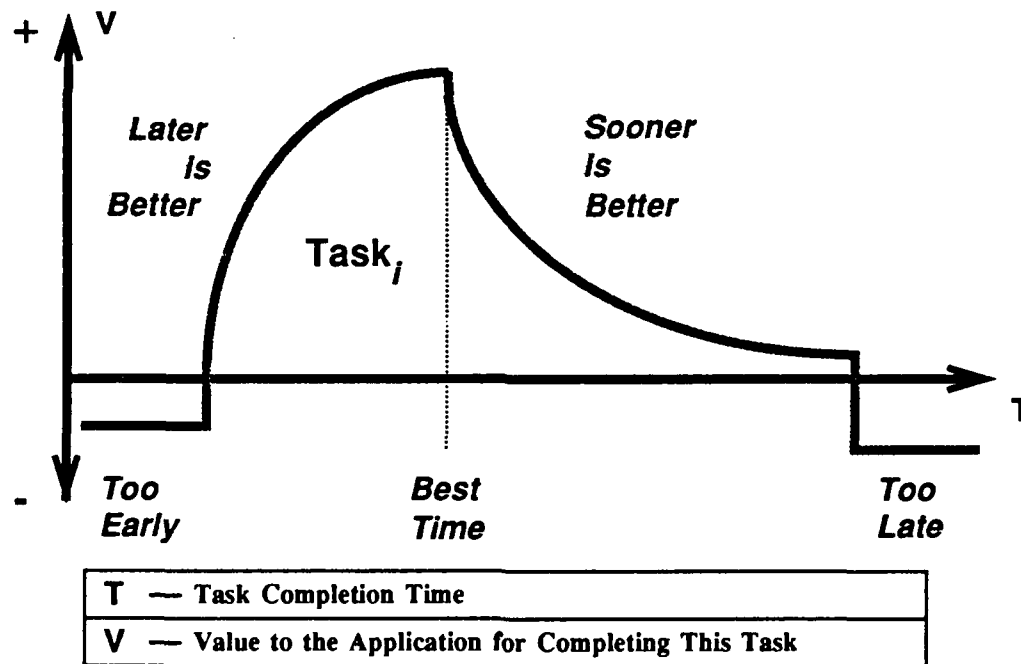
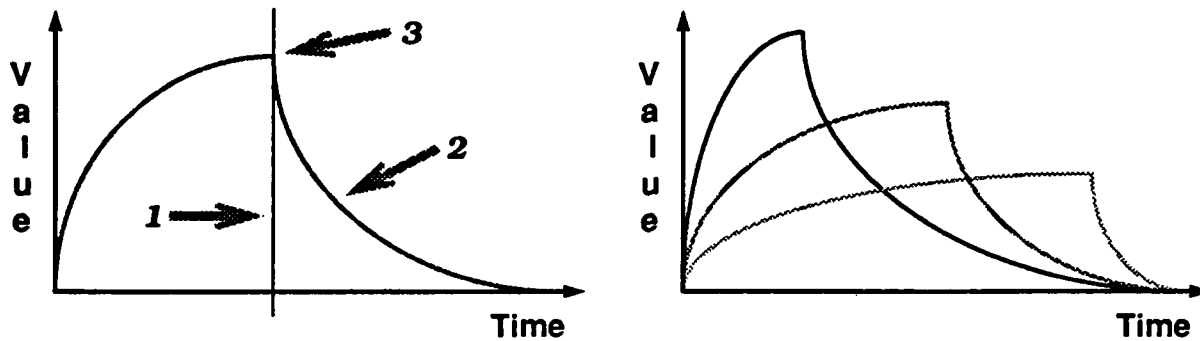


Figure 1. A Time-Value Function

* Jensen invented the concept of time-value functions in the context of scheduling tasks in the Safeguard phased array radar computer system.

hard deadlines as a simple special case (a unit value downward step), and also accommodate a wide variety of soft (i.e., residual value) time constraints. These functions are derived from knowledge of the application's physical nature—e.g., it is counterproductive to launch an interceptor when its target is either out of range or too close. Urgency and importance are dynamic (time- and context-dependent), so in Alpha the functions are allowed to change their parameters at runtime. For example, an interceptor course correction task can have a time-value function which varies in shape as the interceptor approaches its target, as seen in Figure 2.



1. Best time depends on predicted time to intercept
2. Hardness depends on predicted time to intercept
3. Maximum value depends on perceived threat of target

Figure 2. Variable Time-Value Function

Alpha is able to manage all resources, both physical (e.g., processor cycles, primary and secondary storage, i/o and communication) and logical (e.g., tasks, synchronizers such as locks and semaphores, transactions), in a coherent manner using the time-value functions associated with the tasks. The functions for all contending tasks are evaluated collectively, and then the tasks are scheduled so as to maximize the cumulative value to the system for the entire time span they cover; see Figure 3.

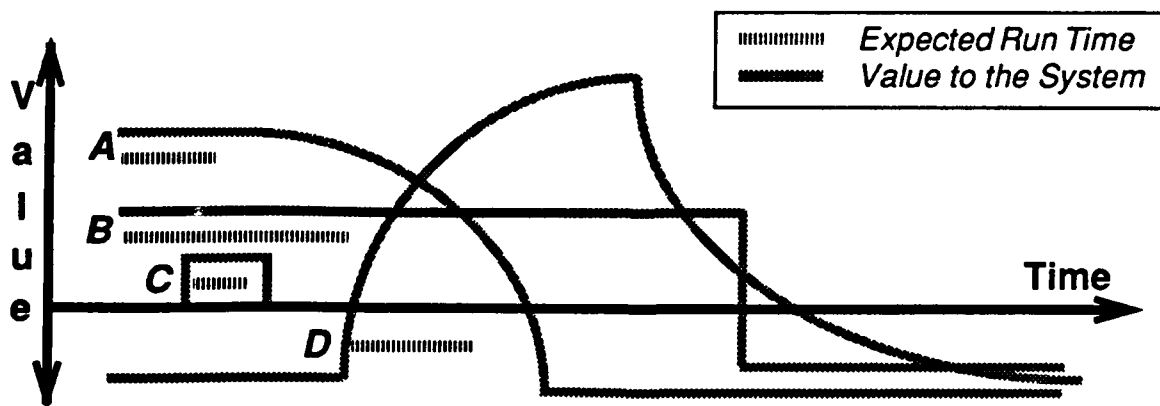


Figure 3. Set of Time-Value Functions

Resource management using time-value functions does exact a price, but it can be high-

ly cost-effective for SIO applications—the resources consumed by the OS for this purpose can yield greater value to the application than if those resources were consumed directly by the application [Trull 88]. In Alpha systems, a large part of that price can be paid in the cheap currency of hardware if desired: a dynamically assigned processor, a dedicated processor, or a special-purpose hardware accelerator (similar to a floating point co-processor) [Jensen 81a].

Time-value functions can also be used to represent “imprecise computations” [Lin 87].

2.2.2 Exceptions

Standard practice in traditional real-time systems is to design and define away as many runtime exceptions as possible for the end of maximally deterministic behavior—e.g., it is usually asserted that no deadline will ever be missed, so no runtime provisions need be made for that eventuality. Standard practice in non-real-time systems is to presume that a relatively large class of anticipated, and perhaps even some unanticipated, runtime exceptions will occur, but that they are unimportant in the sense of not justifying significant resources to deal with them (*cf.* the ubiquitous philosophy of optimizing performance for the most frequent cases, à la the RISC philosophy).

SIO does not have the luxury of either of those standard practices. A large percentage of the runtime exceptions are an integral part of the application itself, and many others are inevitable in dynamic, nondeterministic systems. The OS must be designed to anticipate runtime exceptions yet handle them in a manner which supports the writing of meaningful real-time programs.

One of the most significant kinds of exception in SIO is that it is not only possible, but common, for resource demand to exceed supply; in particular, not all periodic and aperiodic time constraints can always be met. In such cases, application-specific recourse must be taken, and Alpha exploits the task time-value functions to do this. Alpha’s default policy is to gracefully degrade service such that as many as possible of the most important task time constraints are met. Alternative policies could include minimizing the number of missed time constraints, or minimizing the average lateness, or amortizing tardiness across all tasks according to importance.

The performance of any system should be optimized for the most important cases; in SIO, these are often high stress exceptions such as emergencies due to attack or failure, rather than the most frequent (normal, uneventful) cases (as Jensen pointed out in [DRC 86]). Alpha is specifically designed and implemented to exhibit its best performance in critical exception cases. This requires careful reconsideration of programming practices usually taken for granted as valuable. For instance, hints and caches accelerate average performance, and can be counterproductive in real-time if recovery from a wrong hint or cache miss causes unacceptable delay at a critical time. Similarly, one may need to move code *into* loops to expedite exits, and to make remote procedure calls heavier weight in order to facilitate aborts. This approach also suggests the need for related advances in supporting tool technology, such as compiler optimization strategies which accommodate application-specified time- and context-dependent criticality paths (analogous to trace scheduling).

2.2.3 Determinism

It is now understood that a system does not have to *be* perfect to *behave* perfectly (the basis of fault tolerance). Likewise, it should be understood that a system does not have to *be* deterministic to *behave* deterministically. An OS for SIO should behave as predictably as the application actually requires, and is willing to pay for in various ways (such as adaptability and resource supply); but it should not hesitate to employ stochastic means to achieve those ends most cost-effectively—the probability of a long distance telephone call successfully being to its intended destination is very high largely *because of*, not *despite*, dynamic routing in the long distance switching network (and likewise in wide area data communication networks—e.g., [McQuillan 80]). In Alpha, this perspective is manifest most conspicuously in its embrace of aperiodic-based resource management (e.g., scheduling) techniques. Because traditional real-time OS's are focused on low level sampled data subsystems in which periodic tasks are dominant in both number and importance, they can afford the "Procrustean bed" approach to accommodating aperiodic tasks with polling and periodic servers [Lehoczký 87]. But in SIO, time- and mission-critical aperiodic tasks prevail, calling for new resource management principles and techniques. The Alpha project is developing and sponsoring work in this direction, and is taking advantage of related efforts from elsewhere as they develop.

2.2.4 Guarantees

Even in the constrained environment of low level sampled data subsystems, guarantees must be qualified. One sometimes reads of "conditional guarantees" [Biyabani 88], and it is evident that many application users find this oxymoron to be misleading. In the SIO context, the system's exceptional and stochastic behavior implies that guarantees are not only impossible in general, they are conceptually the wrong idea—honoring them would prevent the system from responding to subsequent, more critical needs (in low level, static systems, it is intended that no such subsequent needs could arise). Instead, Alpha performs resource management on a "best effort" basis.

2.2.5 Summary of Real-Time in Alpha

In Alpha, all resources—both logical (e.g., tasks, semaphores, locks, transactions) and physical (e.g., processor cycles, memory pages, i/o bandwidth) are managed directly according to application-specified actual task completion time constraints. Alpha includes aperiodic-based policies which schedule both periodic and aperiodic activities in an integrated, uniform manner, and accommodate dynamic variability and evolution of the time constraints; Alpha also includes a variety of conventional periodic-based policies such as rate-monotonic ones if desired. Alpha supports resource management based on time-value functions, which distinguishes between urgency and importance, and allows not only hard deadlines but also a wide variety of soft (i.e., residual value) completion time constraints. Time-value functions facilitate best effort resource management policies as well as traditional approximations to guarantees. Alpha handles run-time overloads gracefully according to application-specified policies—e.g., meeting as many as possible of the most important time constraints. It optimizes system performance for the most important cases—often high stress exceptions such as emergencies due to attack or failure—not simply for the most frequent (i.e., normal) cases.

Alpha supports the clean-up of computations which fail to satisfy their time constraints,

to avoid wasting resources and executing improperly timed actions. It employs the same block-structured, nested, atomic commit/abort mechanisms for time constraints as it does for transactions.

2.3 Distribution

Distribution is a more difficult issue in mission-oriented real-time systems.

2.3.1 Distributed Operating Systems

In many environments, particularly SIO, the computing system consists of multiple physically dispersed nodes dedicated to performing one distributed application [Franta 81]. For some of these, the loose confederation of systems provided by a conventional computer network is adequate; for others, it is desirable that the network be augmented to present the users at all nodes with a single-system *image*—e.g., so they do not need to know at which node a shared resource is located. But many applications, particularly in SIO, require more than simply the *appearance* of a single system—the physically dispersed nodes must be *actually* integrated into a single system, analogous to a multiprocessor or parallel processor. The kind of distributed OS differs significantly for these three cases.

A conventional distributed (*nee* network) operating system [Thomas 78] is simply a centralized *local* OS with additional standardized interfaces and protocols intended for certain forms of resource sharing among separate applications executing in a network of autonomous nodes—shared file access, remote procedure calls to servers. Usually the sharing is not uniform and transparent to the applications—i.e., the users access remote resources in ways much different from how they access local ones. These interfaces and protocols are typically implemented as utilities at the higher layers of the OS, which minimizes impact of distribution on local OS's, but also limits DOS support for distributed applications. This is illustrated in Figure 4.

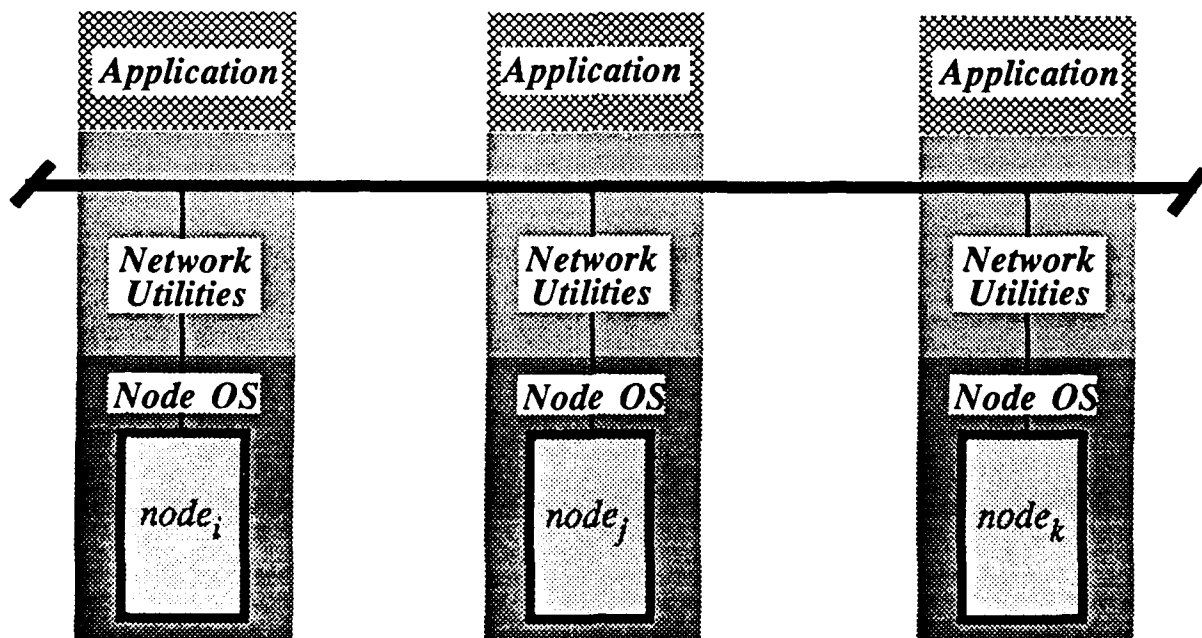


Figure 4. Conventional Distributed Operating System

2.3.2 Distributed Applications

It is often necessary (as in SIO environments) or desirable that an application be distributed across multiple nodes. A distributed application is far more integrated than individual applications which share resources or even interact. It is conceptually a *singular* entity created with a coherent, consistent perspective and set of objectives, not *separate* entities which each have their own independent objectives without concern for, or even awareness of, those of others. Its constituent computations may extend across physical node boundaries for a variety of reasons, rather than each being confined to its own local address space and the resources on one node. The concurrently executing computations *inside* an application exhibit more cooperative behavior than those *among* applications.

2.3.3 Distributed Resource Management

Distributed computations consist of constituent activities (program segments) which exist in an environment of asynchronous, real concurrency of execution; variable, unknown communication delays; and multiple independent node and communication path failure modes. Despite these complexities, it must be possible to coordinate the actions of any number of concurrent activities so that they will have predictable, understandable behavior—i.e., they must explicitly maintain consistency of replicated and partitioned data, and correctness of distributed execution [Jensen 76a]. However, a conventional DOS manages most resources locally per node, and only a few (e.g., for inter-node communication) globally across the whole system. So almost all the distributed resource management for distributed applications must be provided by the users at very high recurring development costs and very high recurring performance penalties—see Figure 5.

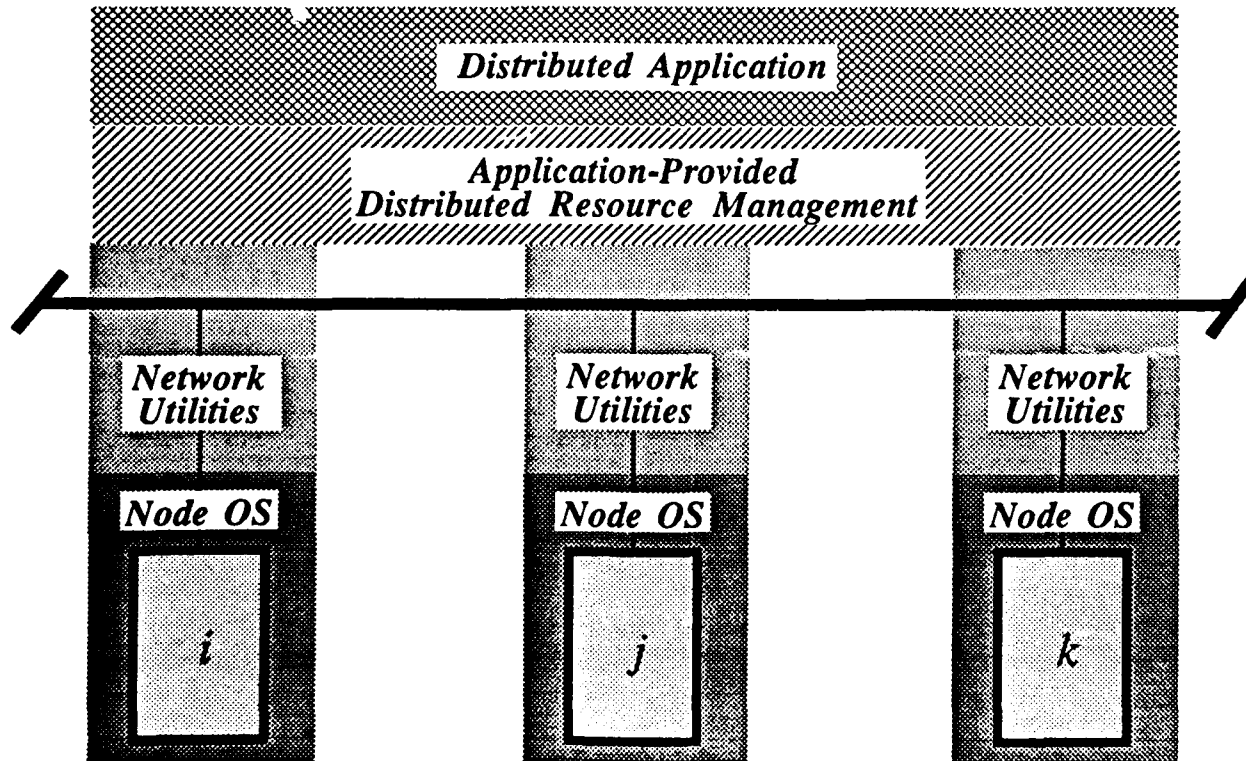


Figure 5. Distributed Application on a Conventional DOS

Distributed resource management should be the responsibility of the system so that the users can devote their efforts to the intrinsic issues of logical structure and algorithms for their applications (see Figure 6); we have used the term “decentralized” for such an OS [Jensen 81b]. Alpha is the first instance we are aware of which qualifies as a decentralized OS.

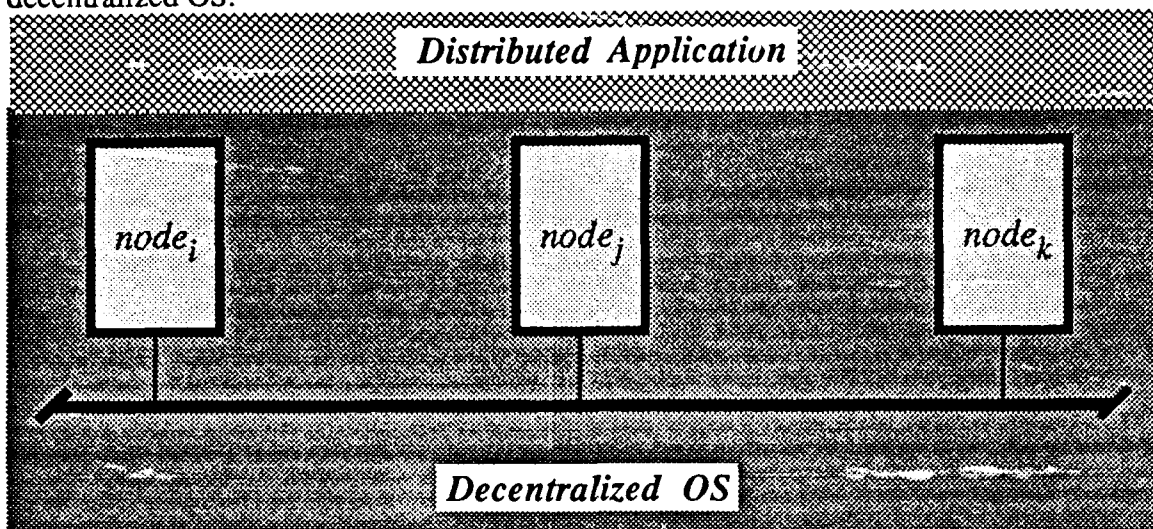


Figure 6. “Decentralized” Operating System

2.4 Survivability

An OS for SIO may have to tolerate conditions far more severe than those encountered in non-real-time contexts. Some (i.e., DoD) systems are subject to overt hostile attack, and so hardware faults tend to be clustered in space and time. Attaining a multi-purpose OS is complicated by the fact that various SIO applications may have a wide variety of mission periods for which there is no single robustness approach, from hours (fighter aircraft) to decades (satellites, industrial plants). Limited or no repairs are possible during the mission. The system usually has to remain in non-stop service during recovery from faults, and there are extreme safety concerns—system failure may jeopardize the mission, human life, and property.

Alpha has four major survivability properties: graceful degradation, fault containment, consistency of data and correctness of actions, and high availability of services and data. Each of these properties is supported by specific kernel-level mechanisms. Graceful degradation is provided by best-effort resource management policies and dynamic reconfiguration of objects. Fault containment results from data encapsulation (objects), having object instances in private address spaces, and capabilities (protected names). Consistency of data and correctness of actions are achieved with concurrency control objects, resource tracking, thread maintenance, block-structured exception handling, and real-time transaction mechanisms. High availability of services and data derives from object replication and dynamic reconfiguration of objects.

Alpha presents its survivability features through the kernel programming model as a set of mechanisms which can be selected and combined as desired—their cost is proportional to their power. The resulting spectrum of survivability includes:

- none
- synchronization operation tracking—prevents deadlock for critical resources
- synchronization operation tracking and exception handlers—allows each individual object instance to satisfy its own consistency constraints
- replication based on synchronization operation tracking and exception handlers—permits an object to survive node failures with nearly up-to-date information
- transactions—scheduled to satisfy application-wide consistency constraints and to simplify the number of failure modes; transaction services and costs may be tailored to suit the needs of the client
- replication based on transactions—permits an object to survive node failures with up-to-date information.

2.5 Adaptability

We consider adaptability to encompass flexibility to accommodate operational variations and modifications at both execution time and configuration time, and evolvability in size (e.g., number of nodes), performance, functionality, and technology. Both of these aspects have special implications on operating systems for SIO.

This environment now demands greater computer system flexibility than has historically been available. Many applications, such as SIO, are becoming so sophisticated that the static approach is clearly infeasible. Also, computer hardware technology advances have diminished the size, weight, and power consumption per unit of performance and the percentage of system costs represented by computer hardware assets, and have increased the cost incentives for standardization and re-usability.

Evolvability presents special challenges in real-time SIO environments because the computing requirements are ill-defined initially and continue to evolve—not just across the design phase, but even across the entire lifetime of the system, and that lifetime can be expected to be decades [Boehm 81].

One of Alpha's most effective forms of support for adaptability is its strict adherence to the philosophy of policy/mechanism separation. It has a kernel of primitive mechanisms from which everything else is constructed according to a wide possible range of application-specific policies to meet particular functionality, performance, and cost objectives. This is essentially the dual of the more widely used principle of information hiding.

Alpha's kernel mechanisms are intended to provide the lowest *meaningful* (not simply the lowest possible) level of functionality for an application—i.e., anything omitted would have to be done by each of the applications, which would result in recurring, inconsistent, inefficient efforts, and thus a much less cost-effective system; anything added would limit policy flexibility and thus system cost/performance tradeoffs. Policy modules written at Alpha's system and user layers may employ the kernel level mechanisms essentially without restrictions—they are “sharp tools.”

Prominent examples of Alpha's policy/mechanism separation include: resource scheduling mechanisms that support best-effort, shortest processing time first, deadline, first-come first-served, or any other policies; and transaction mechanisms (atomicity, concur-

rency control, permanence) that support application-specific transaction policies.

2.6 Programming Model

Alpha's kernel provides a new programming model in support of the technical requirements and approaches discussed above [Northcutt 88b]. The goals for this programming model were that it: explicitly support the timeliness, distribution, survivability, and adaptability objectives needed for SIO; remove the semantic gap between the application's natural abstractions and their more usual system manifestation; be similar in concept and implementation to traditional programming models; eliminate unnecessary scheduler interactions between steps of a computation; not put *a priori* limits on execution concurrency; be oriented towards loosely-coupled architectures (with either uniprocessor or multiprocessor nodes); and not require specialized hardware support for efficient implementation of the basic abstractions.

The principle abstractions of this programming model are:

- objects (passive abstract data types—code plus data), in which there may be any number of concurrent control points
- operation invocation (similar to procedure calling)
- threads (loci of control point execution) which extend through objects via operation invocation.

In addition, Alpha provides real-time distributed data management mechanisms fully integrated into its kernel to achieve the necessary application-specific consistency of replicated and partitioned data, and correctness of distributed execution.

2.6.1 Objects

Alpha's objects are the relatively conventional abstract data type style [Cox 86]. Each instance of an Alpha client-level object has a private address space. The kernel considers the universe of objects to be flat; additional structure is supplied as required at the system and user levels. An instance of an Alpha object exists entirely on a single node. Instances can be dynamically migrated among nodes; initial instance placement is specified by the user. Objects may be replicated transparently, using application-chosen policies. Alpha objects are intended to normally be of moderate size—e.g., 100 to 10,000 lines of code—and number. Everything, including devices and files, appears as objects to the programmer. Alpha's objects are illustrated with a track file object in Figure 7.

Any object may be declared permanent, in the sense that a non-volatile representation of the object's state resides in the object store. Any object may support atomic transaction-controlled updates to its permanent representation, thus providing for consistent, failure-atomic typed data storage. The object store is capable of supporting directory objects to map logical names into object capabilities (which are, themselves, system-provided logical names, and are used exclusively, even within the kernel, except within the object store implementation itself). Should a particular file interface be required, it is possible to create a permanent object class whose interface provides the required file semantics (e.g., operations such as open, close, read, write).

2.6.2 Operation Invocation

Invocation of an operation on an object is the vehicle for all interactions in the system,

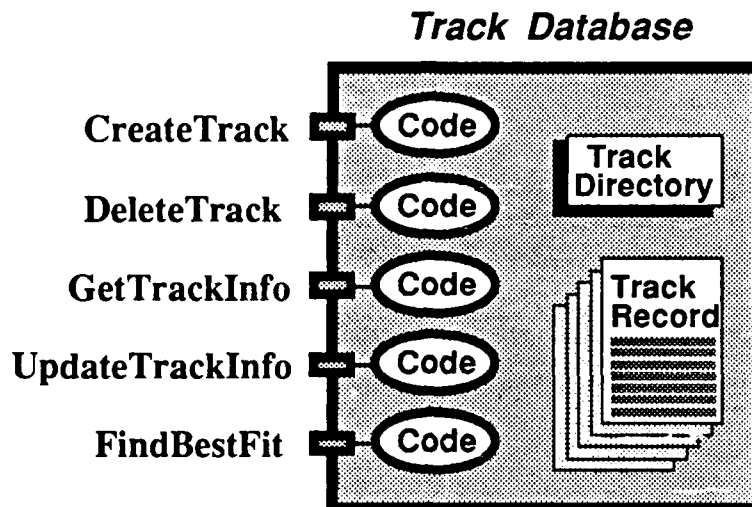


Figure 7. Example Alpha Object

including OS calls. Invocation has synchronous request/reply semantics (similar to remote procedure call [Nelson 81]), but without the nested monitor type limitations often associated with synchronous communications in process/message based systems [Liskov 85b]—operations are block structured.

Invocation masks the effects of physical distribution. Remote objects and object migration provide location transparency. Communication errors are handled by underlying reliable message protocols. Orphan detection and elimination mask node failures.

Invocations may fail for various reasons—e.g., protection violation, bad parameter, node failure, machine exception, time constraint expiration, transaction abort. The kernel provides mechanisms for one block-structured exception handling construct for all these cases; it allows the object programmer to designate application-specific handlers for each type of operation failure, on (if desired) a per-invocation basis.

2.6.3 Threads

An Alpha thread is a distributed computation (not a lightweight task as in Mach [Accetta 86]) which spans both objects and (transparently and reliably) physical nodes, carrying its local state and attributes for timeliness, robustness, etc., as shown in Figure 8. These attributes are used by Alpha at each node to perform resource management on a system-wide basis in the best interests (i.e., to meet the time constraints) of the whole distributed application. The combination of objects and threads is illustrated in Figure 9.

2.6.4 Real-Time Transaction Mechanisms

To achieve the necessary consistency of replicated and partitioned data, correctness of distributed execution, and real-time performance, Alpha's kernel provides transaction mechanisms for failure atomicity, application-specific concurrency control, and permanence. Any combination of these mechanisms may be used, so the particular reliability requirements of an application at a given time can be met for a proportional cost (e.g., execution overhead). This is in contrast to the usual practice of forcing the user to always pay for an entire, heavy-weight transaction facility which he may not require or be able to

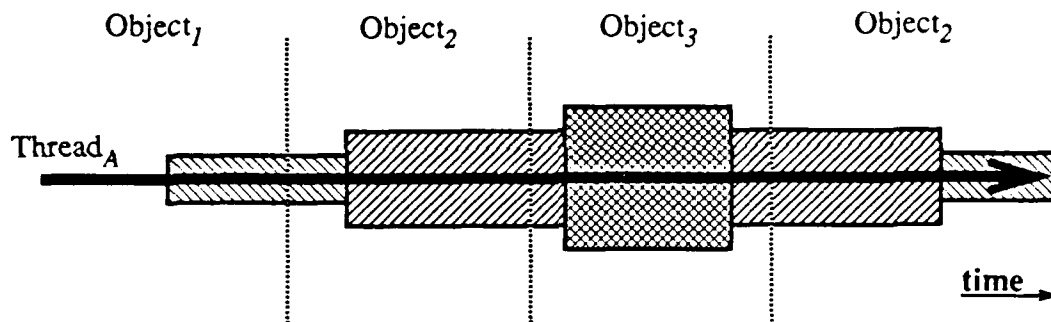
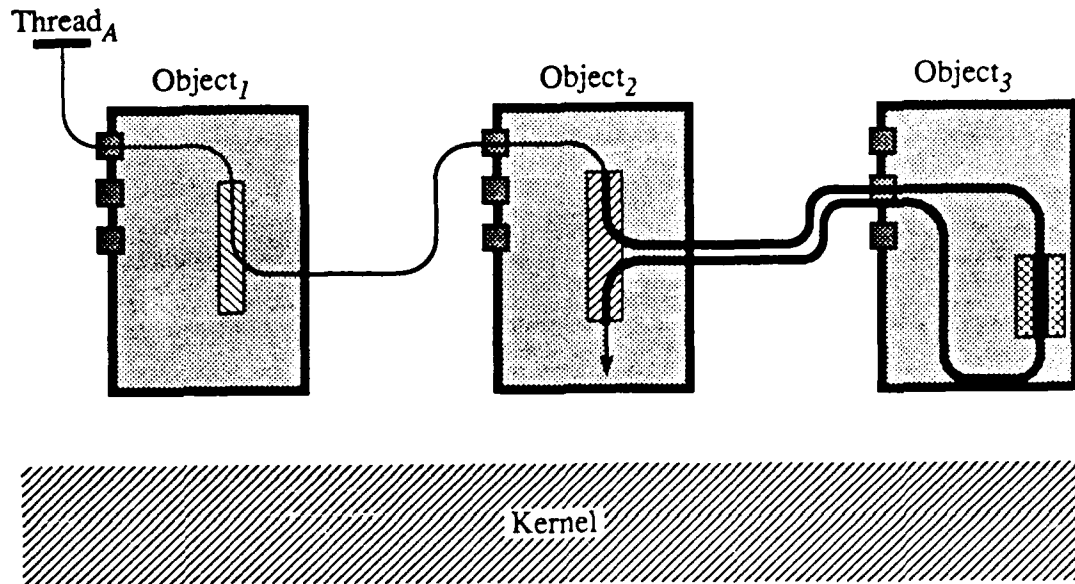


Figure 8. An Alpha Thread and its Attributes

afford. Transactions may be nested, for improved modularity [Moss 85]. These mechanisms are supplied at the kernel level for use not only by the application but also by Alpha, which is itself a set of distributed programs.

Alpha's transactions differ significantly from traditional technology by being *real-time* [Jensen 76a, Abbott 88a, Abbott 88b], which in Alpha basically means that they are scheduled according to the same time-value functions as all other resources at resource request times, and at deadlock resolution times [Clark 89].

Alpha allows not just serializable transactions [Papadimitriou 77], but also non-serializable ones, exploiting various types of knowledge beyond the transaction syntax: the semantics among transactions (e.g., [Allchin 82, Garcia-Molina 83, Liskov 85a]); or the data consistency constraints [Sha 88]. Some forms of non-serializable transactions have advantageous real-time properties with respect to scheduling abort processing—since they commit and allow other transactions to observe their results with no ill effect for any

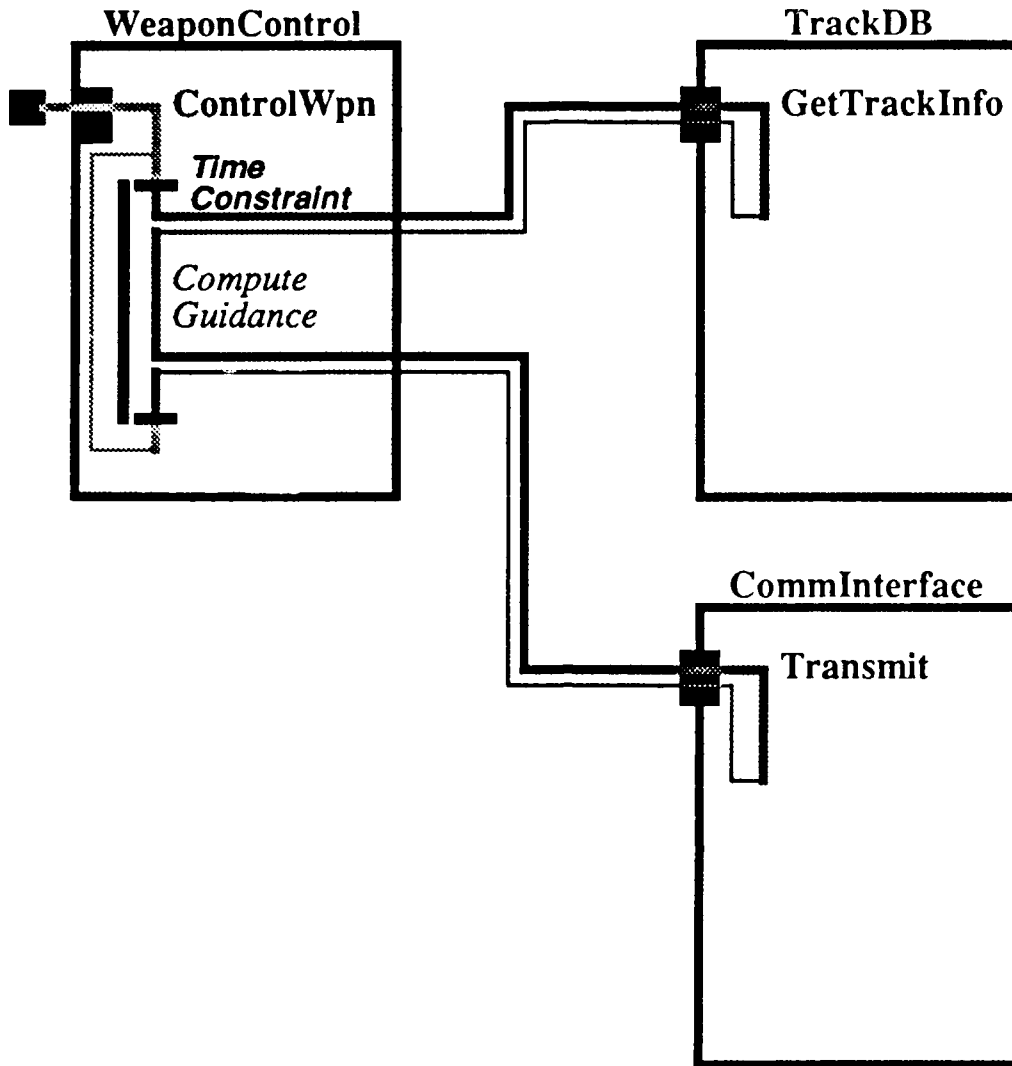


Figure 9. Example of Alpha's Object/Thread Programming Model

arbitrary period of time [Sha 83], their abort processing can also be deferred for an arbitrary period of time (unless there are other mitigating circumstances).

In addition to system-supplied commit and abort handlers for transactions, time constraints, etc., Alpha allows user-supplied handlers customized for application-specific notions of correctness or for improved performance.

2.6.5 Comparison With Other Models

The most common OS programming models are based on the process/message and client/server abstractions.

A process is equivalent to an object and a "captive" thread in Alpha. Processes do not necessarily have well-defined interfaces—they can have entry points at arbitrary places in the code. Many process/message systems do not make use of the generality of asyn-

chronous message passing—they build restricted interfaces on top (*cf.* Matchmaker [Jones 84]). Processes require scheduler interaction and full context swap for each computational step. Process/message systems frequently introduce structure internal to processes (e.g., “light-weight” processes)—this can be considered a loss of correspondence between user and system abstractions. Typically, the distinction between local and remote processes is made explicit—the client must manage logical identifier/location mapping.

The client/server model does not maintain full correspondence between abstraction and implementation of computations. Server processes often execute with their own attributes, independent of the attributes of the client on whose behalf the service is being performed (although a number of OS's do deal with this problem—*cf.* “regions” in iRMX [Intel 78], recently reincarnated as “priority inheritance” in [Sha 87]). The communications and scheduling subsystems (and the interaction between them) dictate the behavior of the system, not the attributes of the computations. Using a single server process to service multiple concurrent requests fails to exploit potential concurrency. Properties of server processes, such as distribution and replication, frequently are not transparent to clients.

Modifications can be made to the process/message and client/server models to better approximate the characteristics of Alpha's thread/object model, however such attempts can be expected to incur substantial costs.

3 System Structure

Figure 10 illustrates the overall structure of the Alpha OS [Northcutt 88f, Northcutt 88g, Northcutt 88h, Jensen 88a, Reynolds 88a, Reynolds 88b]. The mechanisms discussed herein for real-time, distribution, survivability, and adaptability reside in Alpha's kernel. The system layer includes the policies, both system-provided and application-specific. Alpha's user layer is more typical of conventional os's in the sense that it contains the command interpreter, utilities and libraries, and application software.

Releases 2 and beyond of Alpha will co-exist with Concurrent's RTU® [Henize 86] real-time UNIX on each or any node, for software development and access to utilities (such as TCP/IP) whose added value do not warrant their inclusion in Alpha's early releases; it also allows interoperability with BBN's CRONUS wide-area DOS [Schantz 88], which runs on RTU.

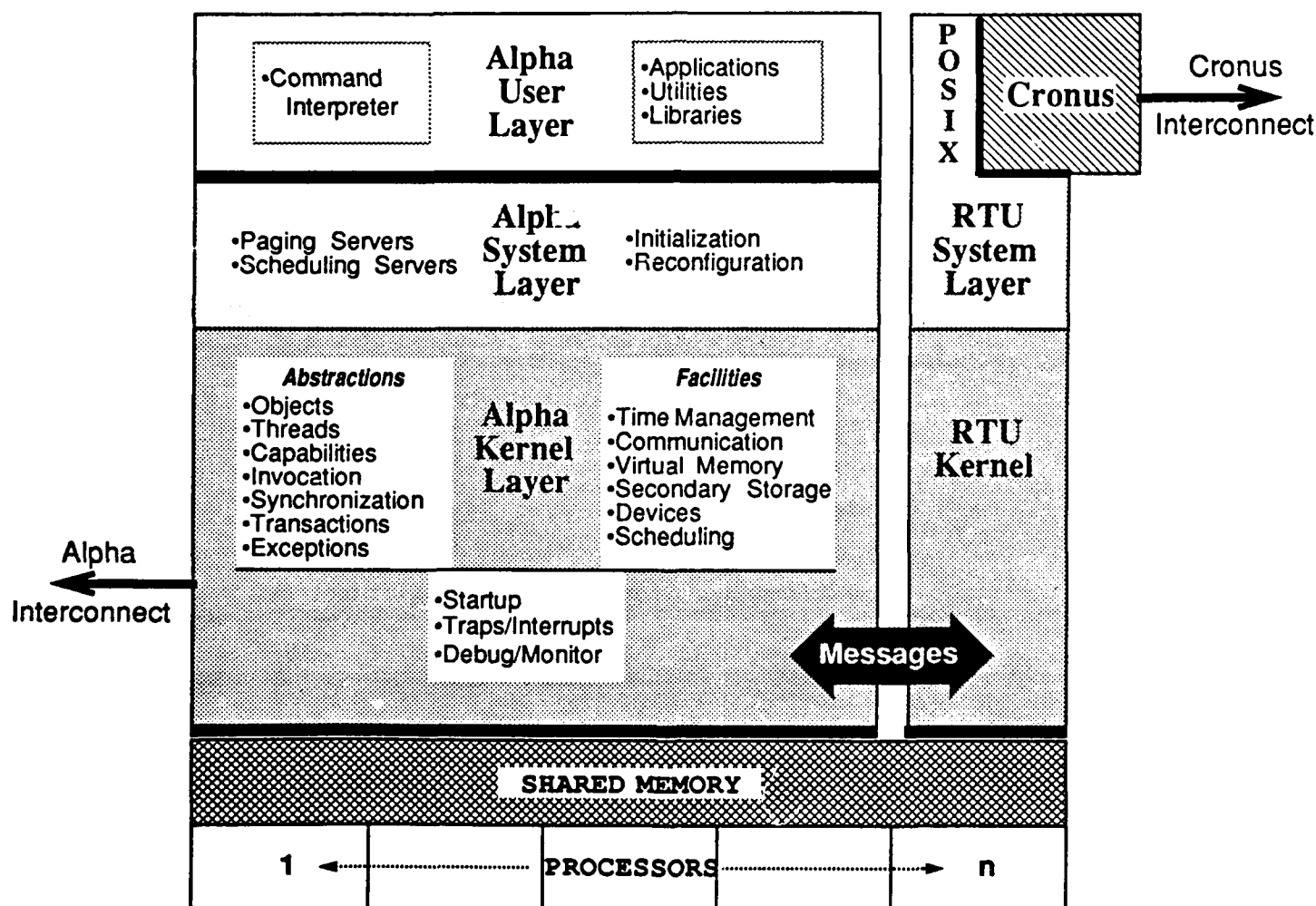


Figure 10. Alpha Releases 2 and 3 Structure

The hardware configuration of the testbed for Alpha Release 1 at CMU and General Dy-

namics can be seen in Figure 11 [Northcutt 88d]. Each multiprocessor node was constructed from modified Sun Microsystems and other Multibus boards.

Development and Control System

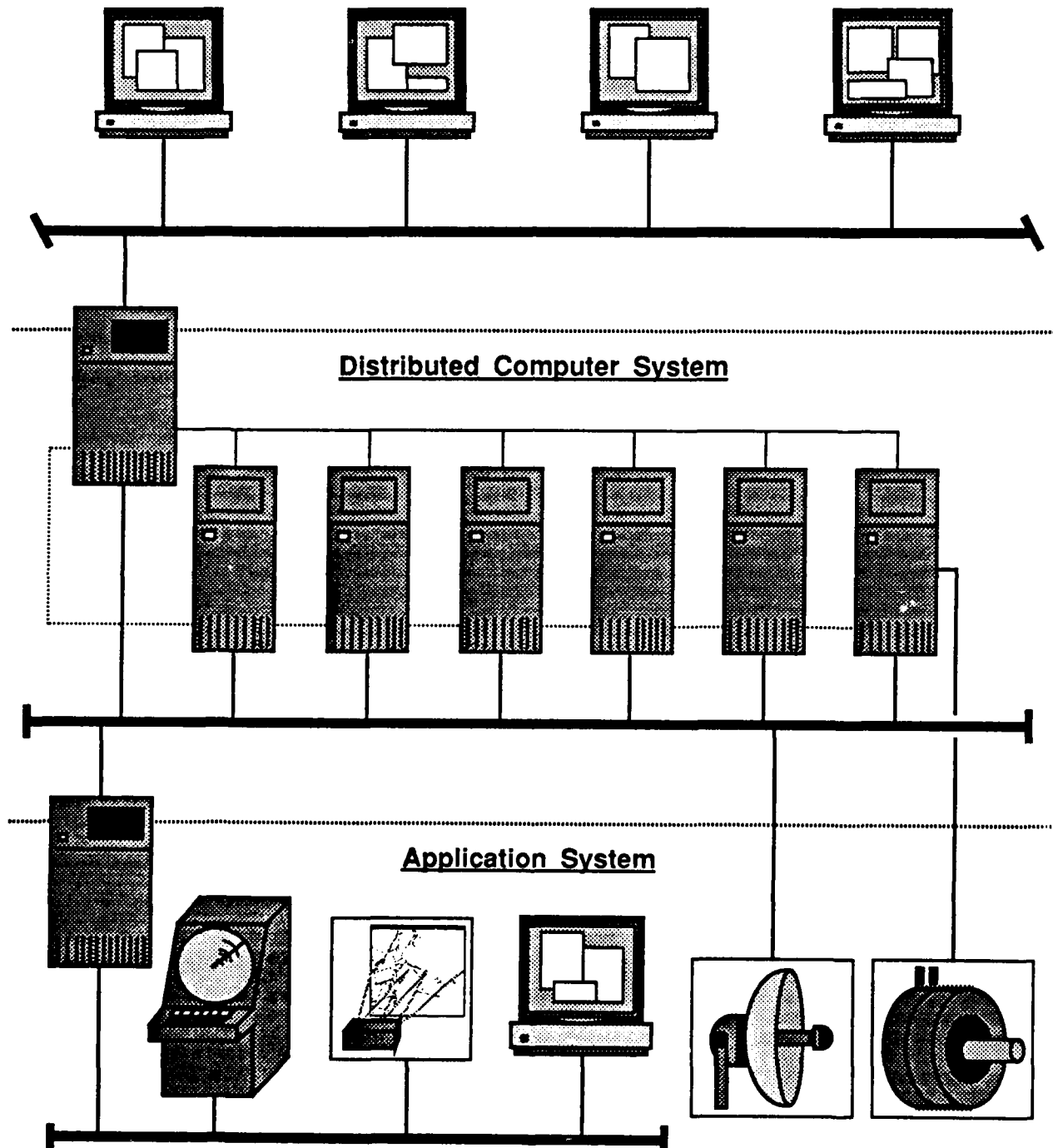


Figure 11. Alpha OS Prototype (Release 1) Testbed

4 Status and Future Plans

The Alpha prototype (now called Release 1) provided validation of its philosophy and basic concepts. However, it was prototype quality, unsupported code on custom-modified and unsupported hardware. Concurrent Computer Corporation is now performing an all-new, next-generation version of Alpha which will respond to lessons learned from Release 1, extend the functionality beyond what the prototype effort had time and resources to provide, and exploit the knowledge of many new contributors. This Alpha will be commercial-quality code, embodied in a phased sequence of increasing functionality releases; major releases 2 and 3 are scheduled for early 1990 and 1991, respectively. This Alpha OS is also portable—its initial hardware platform is a MIPS-based multiprocessor product, and these are then interconnected with FDDI and/or Ethernet—see Figure 12. Alpha OS Releases 2+ and their initial testbed will be installed and supported by Concurrent at a number of Government and industry (both DoD contractor and civilian) facilities, where it will be experimentally evaluated in actual SIO application contexts.

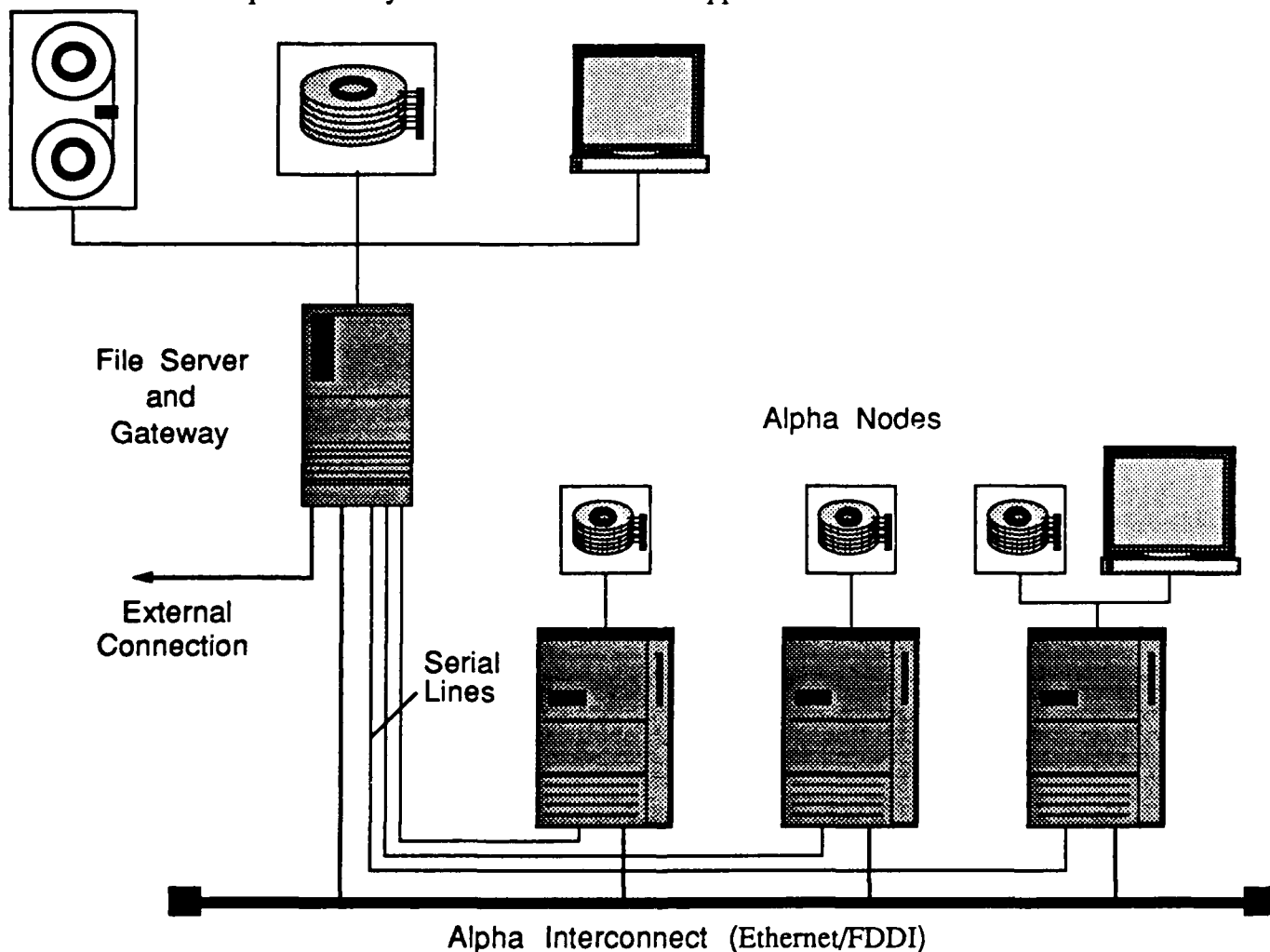


Figure 12. Alpha OS Release 2+ Initial Testbed

Alpha Release 1 was implemented in C; Releases 2 and 3 are being implemented in (AT&T Version 2) C++, and initially will provide runtime library support for C and C++ application software [Shipman 88]. Ada support is being planned for Releases 2 and beyond, and an Ada implementation of the OS is being contemplated.

A number of DoD prime contractors are actively engaged with Concurrent in planning for transition and experimental evaluation of Alpha in applications such as: ground-based battle management, C³I, and air defense; surface ships; and mission management avionics.

Alpha is non-proprietary and in the public domain for U.S. Government uses; both binary and source licenses will be available for commercial purposes on other vendors' hardware platforms.

Alpha enjoys major corporate commitment from Concurrent Computer Corp., both as the basis for a future OS product specifically for the emerging military/aerospace and commercial SIO markets, and as the source of technology for a new generation of real-time UNIX which will be different in kind from commodity UNIX.

5 Acknowledgments

Research on Alpha and its technology, taking place at CMU, Concurrent, and elsewhere, is sponsored in part by the U.S.A.F. Rome Air Development Center; additional support at CMU was provided by the U.S. Naval Ocean Systems Center, and the General Dynamics, IBM, and Sun Microsystems corporations.

The authors are grateful for the contributions to Alpha's design and development by Jim Hanko, Don Lindsay, Martin McKendry, Jack Test, Jeff Trull, and Huay-Yong Wang.

UNIX is a registered trademark of the AT&T Corp. RTU is a registered trademark of Concurrent Computer Corp.

6 References

- [Abbott 88a] Abbott, R. and Garcia-Molina, H.
Scheduling Real-Time Transactions
ACM SIGMOD Record, March, 1988
- [Abbott 88b] Abbott, R. and Garcia-Molina, H.
Scheduling Real-Time Transactions: A Performance Evaluation
Proceedings of the 14th VLDB Conference, ACM, ?, 1988.
- [Accetta 86] Accetta, M. J., Baron, R. X., Golub, D. B., Rashid, R. F., Tevanian, A. and Young, M. W.
Mach: A New Kernel Foundation for UNIX Development.
Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition, June, 1986.
- [Allchin 82] Allchin, J.E., and McKendry, M.S.
Object Based Synchronization and Recovery
Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, 1982.
- [Boehm 81] Boehm, B.W.
Advances in Computer Science and Technology: Software Engineering Economics
Prentice Hall, 1981
- [Biyabani 88] Biyabani, S., Stankovic, J.A., and Ramamritham, K.
The Integration of Deadlines and Criticalness Requirements in Hard Real-Time Systems
IEEE and USENIX Fifth Workshop on Real-Time Software and Operating Systems, May, 1988.
- [Boebert 78] Boebert, W. E.
Concepts and Facilities of the HXDP Executive.
Technical Report 78SRC21, Honeywell Systems & Research Center, March, 1978.
- [Clark 88] Clark, R. K., Kegley, R. B., Keleher, P. J., Maynard, D. P., Northcutt, J. D., Shipman, S. E. and Zimmerman, B. A.
An Example Real-Time Battle Management/Command and Control Application on Alpha.
Archons Project Technical Report #88032, Department of Computer Science, Carnegie-Mellon University, March, 1988.
- [Clark 89] Clark, R.C.
Scheduling Dependent Real-Time Activities
Ph.D. Thesis, Computer Science Department, Carnegie-Mellon University, To Appear.
- [Cox 86] Cox, B.J.
Object-Oriented Programming

- Addison-Wesley, 1986.
- [DRC 86] Dynamics Research Corporation
Distributed Systems Technology Assessment for SDI
Technical Report E-12256U, Electronic Systems Division, USAF
Systems Command, September 1986.
- [Franta 81] Franta, W.R., Jensen, E.D., Kain, R.Y., and Marshall, G.D.
Real-Time Distributed Computers
Advances in Computers, Vol. 20, Academic Press, 1981.
- [Garcia-Molina 83] Garcia-Molina, H.
Using Semantic Knowledge for Transaction Processing in a Distributed Database
ACM Transactions on Database Systems, June, 1983.
- [Henize 86] Henize, J.A.
Understanding Real-Time UNIX
MASSCOMP Technical Report, January, 1986.
- [Intel 78] Intel Corp.
iRMX86 Nucleus Users Guide
Intel Corporation, 1978
- [Jensen 76a] Jensen, E. D.
The Implications of Physical Dispersal on Operating Systems.
Workshop on Distributed Processing, Brown University, Providence, RI, August, 1976.
- [Jensen 76b] Jensen, E. D. and Anderson, G. A.
Feasibility Demonstration of Distributed Processing for Small Ships Command and Control Systems.
Final Report N00123-74-C-0891, Honeywell Systems & Research Center, August, 1976.
- [Jensen 79] Jensen, E. D.
Distributed Computer Systems
Computer Science Research Review, Department of Computer Science, Carnegie-Mellon University, September, 1979.
- [Jensen 81a] Jensen, E.D.
Hardware/Software Relationships in Distributed Computer Systems
Distributed Systems—Architecture and Implementation: An Advanced Course, Springer-Verlag, 1981.
- [Jensen 81b] Jensen, E.D.
Decentralized Control
Distributed Systems—Architecture and Implementation: An Advanced Course, Springer-Verlag, 1981.
- [Jensen 84] Jensen, E. D., and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System—An Overview of

- its Objectives and Approach*
IEEE Distributed Processing Technical Committee Newsletter,
Special Issue on Distributed Operating Systems, June, 1984.
- [Jensen 85] Jensen, E.D., Locke, C.D., and Tokuda, H.
A Time-Driven Scheduling Model for Real-Time Operating Systems
Proc. IEEE Real-Time Systems Symposium, December, 1985.
- [Jensen 88a] Jensen, E.D., Test, J.A., Reynolds, F.D., Burke, E., Hanko, J.G.
Alpha Release 2 Design Summary Report.
Technical Report #88091, Kendall Square Research Corporation,
September 1988.
- [Jensen 88b] Jensen, E. D.
Alpha: A Real-Time Decentralized Operating System for Mission-Oriented System Integration and Operation
Proc. Symposium on Computing Environments for Large, Complex Systems, Univ. of Houston Research Institute for Computer and Information Sciences, November 1988.
- [Jensen 88c] Jensen, E. D., Northcutt, J. D., Clark, R. K., Shipman, S. E., Maynard, D. P. and Lindsay, D.C.
The Alpha Operating System: An Overview.
Archons Project Technical Report #88121, Department of Computer Science, Carnegie-Mellon University, December, 1988.
- [Jones 84] Jones, M. B., Rashid, R. F., and Thompson, M. R.
Matchmaker: An Interface Specification Language for Distributed Processing.
Technical Report CMU-CS-84-161, Department of Computer Science, Carnegie-Mellon University, 1984.
- [Lehoczky 87] Lehoczky, J.L., Sha, L., and Strosnider, J.K.
Enhanced Aperiodic Responsiveness in Hard-Real-Time Environments
Proc. IEEE Real-Time Systems Symposium, 1987.
- [Lin 87] Lin, K. J., Natarajan, S., and Liu, J.W.S.
Imprecise Results: Using Partial Computations in Real-Time Systems
Proc. IEEE 8th Real-Time Systems Symposium, December 1987.
- [Liskov 85a] Liskov, B. and Weihl, W.
Specification of Distributed Programs
Technical Report, MIT, September, 1985.
- [Liskov 85b] Liskov, B. H., Herlihy, M. P. and Gilbert, L.
Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing.
Technical Report CMU-CS-85-168, Department of Computer Science, Carnegie-Mellon University, October, 1985.

- [Liu 73] Liu, C.L. and Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment
Journal of the ACM, 20(1), 1973.
- [Locke 86] Locke, C.D.
Best-Effort Decision Making for Real-Time Scheduling
Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1986.
- [McQuillan 80] McQuillan, J.M., Richer, I., and Rosen, E.C.
The New Routing Algorithm for the ARPANET
IEEE Transactions on Communications, May, 1980
- [Moss 85] Moss, J.E.B
Nested Transactions: An Approach to Reliable Distributed Computing
MIT Press, 1985.
- [Nelson 81] Nelson, B. J.
Remote Procedure Call.
Ph.D. Thesis. Department of Computer Science, Carnegie-Mellon University, May, 1981.
- [Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.
- [Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January, 1988.
- [Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie-Mellon University, February, 1988.
- [Northcutt 88c] Northcutt, J. D., Clark, R. K., Shipman, S. E., Maynard, D. P., Lindsay, D. C., Jensen, E. D., Smith, J. M., Kegley, R. B., Keleher and Zimmerman, B. A.
Alpha Preview: A Briefing and Technology Demonstration for DoD.
Archons Project Technical Report #88031, Department of Computer Science, Carnegie-Mellon University, March, 1988.
- [Northcutt 88d] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.

- [Northcutt 88e] Northcutt, J. D. and Shipman, S. E.
The Alpha Operating System: Programming Utilities.
Archons Project Technical Report #88041, Department of Computer Science, Carnegie-Mellon University, April, 1988.
- [Northcutt 88f] Northcutt, J. D., Clark, R. K., Shipman, S. E. and Lindsay, D. C.
The Alpha Operating System: System/Subsystem Specification.
Archons Project Technical Report #88051, Department of Computer Science, Carnegie-Mellon University, May, 1988.
- [Northcutt 88g] Northcutt, J. D.
The Alpha Operating System: Kernel Programmer's Interface Manual.
Archons Project Technical Report #88061, Department of Computer Science, Carnegie-Mellon University, June, 1988.
- [Northcutt 88h] Northcutt, J. D. and Shipman, S. E.
The Alpha Operating System: Program Maintenance Manual.
Archons Project Technical Report #88071, Department of Computer Science, Carnegie-Mellon University, July, 1988.
- [Papadimitriou 77] Papadimitridriou, C.H., Bernstein, P.H., and Rothnie, J.B.
Some Computational Problems Related to Database Concurrency Control
Proc. Conference on Theoretical Computer Science, August, 1977.
- [Peng 87] Peng, D. and Shin, K.G.
Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control
IEEE Transactions on Computers, April, 1987.
- [Ready 86] Ready, J.F.
VRTX: A Real-Time Operating System for Embedded Microprocessor Applications
IEEE Micro, August, 1986.
- [Reynolds 88a] Reynolds, F.D., Hanko, J.G., Test, J.A., Burke, E., Jensen, E.D.
Alpha Release 2 Kernel Interface Specification
Technical Report #88121, Concurrent Computer Corporation, December 1988.
- [Reynolds 88b] Reynolds, F.D., Hanko, J.G., Jensen, E.D.
Alpha Release 2 Preliminary System/Subsystem Description
Technical Report #88122, Concurrent Computer Corporation, December 1988.
- [Schantz 88] Schantz, R.E. and Thomas, R.H.
Cronus, A Distributed Operating System: Functional Definition and System Concept
Technical Report RADC-TR-88-80, Rome Air Development Cen-

- ter, 1988.
- [Sha 83] Sha, L., Jensen, E.D., Rashid, R.F., and Northcutt, J.D.
Distributed Co-Operating Processes and Transactions
Synchronization, Control, and Communication in Distributed Computing Systems, Academic Press, 1983.
- [Sha 87] Sha, L., Rajkumar, R., Lehoczky, J.P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization
Technical Report CMU-CS-87-181, Department of Computer Science, Carnegie-Mellon University, 1987.
- [Sha 88] Sha, L., Lehoczky, J.P., and Jensen, E.D.
Modular Concurrency Control and Failure Recovery
IEEE Transactions on Computers, 1988.
- [Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, October, 1988.
- [Stankovic 87] Stankovic, J.A. and Sha, L.
The Principle of Segmentation for Hard Real-Time Systems
Technical Report, 1987
- [Stankovic 88] Stankovic, J.A., and Ramamritham, K.
Hard Real-Time Systems
IEEE Computer Society Tutorial, 1988.
- [Stewart 77] Stewart, B.
Distributed Data Processing Technology, Interim Report to the U.S. Army Ballistic Missile Defense Advanced Technology Center
Honeywell Systems and Research Center, March, 1977.
- [Thomas 78] Thomas, R.H., Schantz, R.E., and Forsdick, H.C.
Network Operating Systems
Technical Report 3796, Bolt, Beranek and Newman, 1978.
- [Trull 88] Trull, J. E., Northcutt, J. D., Clark, R. K., Shipman, S. E. and Lindsay, D. C.
An Evaluation of the Alpha Real-Time Scheduling Policies.
Archons Project Technical Report #88102, Department of Computer Science, Carnegie-Mellon University, October, 1988.

Table of Contents

Abstract	B-1
1 Introduction	B-4
2 Application Domain	B-6
2.1 Distributed Systems	B-6
2.2 Real-Time Command and Control	B-8
3 Special Requirements	B-11
3.1 Timeliness	B-11
3.1.1 Standard Requirements	B-11
3.1.2 Time-Driven Resource Management	B-12
3.1.3 Common Misconceptions	B-13
3.1.3.1 Priority-Based Processor Scheduling	B-13
3.1.3.2 Deterministic Systems	B-14
3.1.3.3 Excess Assets	B-16
3.2 Distribution	B-16
3.3 Robustness	B-17
3.3.1 Implications of Distributed Real-Time Control	B-17
3.3.2 Major Robustness Concepts	B-18
3.3.2.1 Correctness of Data and Actions	B-18
3.3.2.2 Availability of Services and Data	B-18
3.3.2.3 Graceful Degradation of Function	B-19
3.3.2.4 Fault Containment	B-19
3.3.3 Optimizing for Exception Cases	B-20
3.4 Adaptability	B-20
4 Current Practice	B-22
4.1 Timeliness	B-22
4.1.1 Minimalist Systems	B-22
4.1.2 Priority-Based Scheduling	B-23
4.1.3 Low Utilization	B-24
4.2 Distribution	B-24
4.3 Robustness	B-26
4.4 Adaptability	B-27
4.4.1 Dynamic System Behavior	B-27
4.4.2 Dynamic Time Constraints	B-28
5 Technical Approach	B-30
5.1 Basic Abstractions	B-31
5.1.1 Objects	B-31
5.1.2 Threads	B-33
5.1.3 Operation Invocation	B-35
5.2 Time-Value Functions	B-37
5.3 Implementation Features	B-39

5.3.1	System Software Structure	B-39
5.3.2	Testbed Architecture	B-41
6	Rationale	B-43
6.1	Timeliness	B-43
6.1.1	Effects on the Basic Abstractions	B-43
6.1.1.1	The Thread/Object Approach	B-44
6.1.1.2	Comparisons to Traditional Approaches	B-44
6.1.2	Effects on the Programming Interface	B-48
6.1.2.1	Specifying Timeliness Attributes	B-48
6.1.2.2	Handling Expired Time Constraints	B-49
6.1.3	Effects on the Kernel Subsystems	B-51
6.1.3.1	Scheduling Subsystem	B-52
6.1.3.2	Communications Subsystem	B-54
6.1.3.3	Storage Management Subsystem	B-54
6.1.4	Effects on the Kernel Mechanisms	B-55
6.2	Distribution	B-56
6.3	Robustness	B-57
6.3.1	Exception Handling	B-58
6.3.2	Optimizing for Exceptions	B-60
6.4	Adaptability	B-61
6.4.1	Object-Oriented Programming	B-61
6.4.2	Policy/Mechanism Separation	B-62
8	Acknowledgments	B-63
	References	B-64

List of Figures

Figure 1	Example Decentralized System	B-6
Figure 2	Example Distributed System	B-7
Figure 3	Real-Time Supervisory Control Context.....	B-9
Figure 4	Example Real-Time Systems Hierarchy.....	B-10
Figure 5	Static versus Dynamic Priorities.....	B-15
Figure 6	Optimizing for Exception Cases.....	B-20
Figure 7	Example Object	B-32
Figure 8	Example Thread/Object Snapshot	B-34
Figure 9	Example of Thread Attribute Nesting	B-35
Figure 10	Example Operation Invocation.....	B-36
Figure 11	Components of a Time-Value Function	B-38
Figure 12	Logical System Software Structure	B-40
Figure 13	Testbed System Structure	B-42
Figure 14	Typical Process/Message Interactions.....	B-46
Figure 15	Typical Thread/Object Interactions	B-47
Figure 16	Example Use of Time Constraint Blocks	B-51

Abstract

Alpha is an adaptable decentralized operating system for real-time applications, being developed as a part of the Archons project's on-going research into real-time distributed systems. Alpha is a new kind of operating system, which is unique in two highly significant ways—first, it is *decentralized*, providing reliable resource management transparently across physically dispersed nodes, so that distributed applications programming can be done as though it were centralized; and second, it provides comprehensive high technology support for real-time applications, particularly supervisory control systems (e.g., industrial automation) which are characterized by predominately aperiodic activities, having critical time constraints (such as deadlines) associated with them. Alpha is extremely adaptable so as to be easily optimized for a wide range of problem-specific functionality, performance, and cost.

Alpha is oriented towards systems having on the order of 10 to 100 nodes, which are physically dispersed on the order of 1 to 1000 meters. The Alpha operating system is for the most demanding kind of situation: mission-oriented systems where all nodes are contributing to the same application, not simply for the network case of individual users at each node doing unrelated computations. The focus of this research is on having nodes be logically integrated together, rather than autonomous. Alpha provides this logical integration by executing on the bare hardware and managing resources in the same sense as a uniprocessor operating system does, not by being just a "UNIX-style" user process and providing standard application interfaces and protocols for simple inter-node resource sharing, as is done in conventional, computer network-style distributed operating systems. Resources must often be managed by Alpha across node boundaries in the best interests of the whole application, not just on the usual per-node basis. This necessitates that Alpha also accept responsibility for handling asynchronous concurrency and reliability issues which arise in distributed systems, instead of passing them all up to the users for recurring, lower performance solutions. Alpha provides facilities which are necessary and sufficient to maintain consistency of data and correctness of operation at both the system and application levels, despite concurrent execution and node or communication path failures. This is accomplished through the use of techniques similar to those normally found, far above the operating system, in distributed database systems—e.g., nested atomic transactions, replication. With Alpha, the system's constituent nodes collectively form a single computer, not a computer network; thus, distributed application software can be written as though it were for a conventional uniprocessor—without the applications programmer even knowing about, much less having to manage, distributed resources.

The term "real-time" is usually intended to mean "deterministic behavior" and "faster is better," particularly in the area of interrupt handling and context swaps. Real-time control in this sense applies only to computer systems which simply do low-level sensor/actuator sampled-data loop applications, and are traditionally designed to have rigidly periodic behavior. But supervisory real-time control is far more difficult because it encompasses not just such static periodicity but also predominantly dynamic and aperiodic activities which nonetheless have critical time constraints, such as deadlines. These constraints are part of the correctness criteria of the computation, and failure to meet them is a threat to the systems's mission and to survival of property and human life.

A novel approach is taken in Alpha whereby the time constraints of each of an application's constituent activities are expressed in terms of the value to the system of completing each activity as a function of its completion time (deadlines are a simple special case of this, that is, a unit value step function). In addition, computational activities have relative importances which are also time-dependent. These time-value functions and importances are dynamic and are used by the system in the time-driven management of system resources. The conventional and seemingly simpler notions of "priority" in real-time systems are poor approximations to this approach, and extensive experience has consistently demonstrated that priorities introduce massive and uncontrollable complexity into all but the most trivial real-time systems. Alpha employs this new real-time management technique to resolve *all* contention for resources such as processor cycles, communication access, secondary storage, and synchronizers (e.g., semaphores and locks). Time constraints and importance are among the attributes propagated with computations which cross node boundaries so that resource management can be global. The ubiquitous client/server model is unsuitable in this respect since it does not maintain such essential correspondences between the service and client on whose behalf that service is being provided.

Furthermore, Alpha exhibits a fundamental philosophy which is contrary to that of operating systems for other application environments. Instead of optimizing performance of the normal cases at the expense of infrequent ones, it does the opposite. It is in the exception cases such as emergencies (e.g., being in danger due to failure or attack) when a real-time operating system must be depended upon to perform best, even if the system's routine performance must be suboptimal to ensure that. This is one of the principal reasons why the expression "real-time UNIX" is inevitably an oxymoron.

Supervisory real-time control applications are very complex, and are not (perhaps cannot be) well understood; in addition, the environment and technology are always in a state of flux. Thus, the functional and performance requirements for their computers evolve continuously throughout the entire life cycle of the system (which can be decades). Alpha accommodates this situation through a variety of techniques, many of which are quite innovative. Its design is kernelized and strictly adheres to the principle of policy/mechanism separation. Specific operating system policies are carefully excluded from its kernel level mechanisms so that a wide range of different service facilities, and indeed entire decentralized operating systems, can be effectively constructed using Alpha's kernel, in accordance with application needs. For example, Alpha's kernel provides atomicity, serializability, and permanence of operations as orthogonal mechanisms. Conventional atomic transaction facilities bundle all three properties together, with correspondingly high overhead, as the only choice of policy regardless of need and affordability. But the client layers of Alpha's kernel can base their policies on other combinations of these mechanisms. For example, there are many instances in real-time systems when problem-specific consistency constraints yield correct results more efficiently than serializability would, or when permanence is not worth its cost. This same philosophy is followed in scheduling, communications, and all other types of system resource management.

Computers embedded in real-time control systems usually must produce the highest possible performance from the allowable hardware size, weight, and power, including

memory space for the operating system. A general-purpose computer system can easily be an order of magnitude lower in performance than a special-purpose one for a particular application. Thus, to achieve the balance of performance and flexibility needed for cost-effectiveness in a multiplicity of changing real-time applications, Alpha is general-purpose but unusually malleable so as to exploit all the problem-specific static and dynamic information available from the application. In addition, application functionality can readily be migrated downward into the operating system, and even into its kernel, for increased performance when necessary.

Alpha's internal implementation is organized so that its subsystems (such as scheduling, communications, secondary storage, etc.) can all execute truly concurrently within each node. It is intended that these separate hardware points of control within Alpha be a mixture of dynamically assigned general-purpose processors (i.e., each node in the decentralized computer can be a multiprocessor) and algorithmically specialized hardware accelerators (co-processors and other forms of augmentation). Alpha extends to its client applications the same opportunities for taking advantage of multiple special-purpose as well as general-purpose processors at each node.

Alpha presents a programming model which is object-oriented, in the sense of supporting basic programming units which adhere to the definition of abstract data types. This imposes a structure and discipline conducive to modular software at both the operating system and application levels, as well as improving fault isolation. The active entity, or unit of logical computation, is known as a *thread*, whose execution progresses through objects via operation invocation, without regard for address spaces or node boundaries; distribution and reliability are the responsibility of Alpha instead of the user. This network uniformity and transparency greatly aid the creation and modification of distributed applications.

Alpha is the first systems effort of the Archons Project, and the prototype was created at Carnegie-Mellon University directly on multiprocessor nodes constructed with Sun workstation hardware [Northcutt 88b]. It has been demonstrated with a real-time control application written by its first industrial user, General Dynamics [Maynard 88]. An enhanced, second-generation, commercial-quality version being produced at Kendall Square Research is portable but initially targeted at one of their multiprocessor products. Both versions of Alpha are sponsored by the USAF Rome Air Development Center and are in the public domain for U.S. Government use.

1 Introduction

The overall goals of the Archons project involve the exploration of both architectural and operating system issues related to real-time command and control. The Archons project began following a major hardware construction effort (i.e., the HXDP [Jensen 78a] system constructed at Honeywell's Systems and Research Center), is now in the midst of an operating system construction effort, and plans are being made for the next phase of hardware construction.

The research described in this document is based on a significant amount of theoretical study and substantial experience with a variety of actual real-time command and control problems and systems. The Alpha operating system is being developed as the first implementation effort of the Archons project. This empirical effort is an important phase in the project's overall "think-do" strategy—i.e., a research approach based on a repeating cycle of conceptualizing and theorizing, designing and implementing, then measuring and evaluating.

The Archons work is focused on pushing in new directions to achieve greater benefits than is available by obtaining the next increment from some aging technology. Alpha is not merely an academic exercise, but a validation of the project's conceptual efforts. Real-time command and control is an area that is well understood by the members of the Alpha team (which represents over 70 years of collective experience in the real-time area), as is the need for research to advance the state of the art in the area. Alpha is directed more towards meeting the needs of this problem area than simply generating academic publications.

Real-time command and control applications demand a new kind of operating system which directly addresses their special requirements in the areas of: meeting user-specified real-time processing constraints, supporting complex distributed applications, survivability under hostile attack, and adaptability to evolving needs across (frequently decade-long) system lifetimes. These characteristics all demand that the operating systems needed for real-time command and control differ drastically, not just in degree but in kind, from conventional ones, calling for dramatically innovative concepts and techniques. Very little technology has, or will, come from the mainstream academic and industrial operating system research and development community, which is primarily focused on completely different types of applications (e.g., personal productivity workstations, throughput-oriented numerically intensive computing, and time-independent transaction processing).

The real-time command and control application area has vital requirements which strongly differentiate it from the context in which almost all academic research and commercial product development on operating systems takes place. First, most of the activities in the system (and therefore in the computer) have stringent time constraints which are a matter of correctness rather than convenience, and responsiveness to these constraints, not throughput, is the primary metric of performance. Second, the safety of human life and property depends on the survivability of the system, despite its being subjected to warfare. Third, even though the system consists of physically dispersed elements, they are all dedicated to cooperatively performing a single mission, instead of being simply a network of communicating but otherwise independent and unrelated individual entities. Finally, the system must be adaptable and evolvable to accommodate

significant changes in the mission, the environment, and the technology base across a period of decades, often while remaining in non-stop service.

Another crucial, but often overlooked, factor in achieving the most cost-effective real-time operating system is that the performance of the system must be optimized for the high-stress exception cases, such as emergencies caused by hostile attack or faults, even if the normal case performance must be suboptimal to accomplish that. This is diametrically opposed to the prevailing approach in non-real-time operating system design and implementation, where performance is optimized for the most common case at the expense of less frequent cases.

The programming model that was developed in this effort stems from an examination of the requirements of the problem area, the in-depth study of several key techniques for solving the fundamental problems, and the application of sound engineering trade-offs in combining the individual features into a coherent system. This is different from other operating system efforts in a number of ways. The design of Alpha was derived top-down, from high-level requirements and not bottom-up, from a set of implementation details. Most significantly, Alpha was not constrained to be compatible with existing (system or application) software, and so the exploration of the major research concepts was not compromised by having to splice the newly developed features into an existing system.

The Alpha programming model was derived from the requirements defined for the Alpha operating system's chosen application domain—i.e., distributed real-time command and control applications. There is a unique combination of requirements implied by this application domain, and existing systems either do not attempt to meet these requirements at all, or do not do so in a comprehensive, well-integrated fashion. The Alpha programming model provides features that meet each of the main requirements of the defined applications domain.

This is a technical overview of the Alpha operating system, with emphasis on the features of systems research that are frequently overlooked, such as the context within which Alpha is being created, and the assumptions upon which this work is based. In this document, the Alpha operating system's intended application domain is defined, and the special requirements of the chosen problem area are described. This report goes on to motivate the project's work by exploring the ways in which existing operating systems do not adequately address the needs of this area, and the technical approach taken in Alpha is presented in brief. Finally, some of the rationale for the design of the Alpha system is given and specific features of the system are related back to the defined requirements.

2 Application Domain

This section describes the application domain from which the Alpha operating system's requirements were derived. The intent of this description is to explicitly define the assumptions upon which this work has been done, in order to provide a basis upon which to state the rationale for the Alpha operating system and to provide a context in which the work can be judged.

2.1 Distributed Systems

Because of the widely differing definitions of "distributed systems" that abound in the literature, it is important that the definition used in this document be made explicitly clear. For the purposes of this research, a *decentralized computer* is considered to be a machine that consists of a multiplicity of physically dispersed processing nodes, integrated into a single computer through a native, global *decentralized operating system* [Jensen 78a] (as shown in Figure 1). A decentralized operating system manages the system's collective, disjoint physical resources in a unified fashion, for the common good of the whole system. This permits a distributed system's disjoint resources to be applied directly to a single application.

A logically singular, yet physically dispersed computer is highly appropriate in many contexts (e.g., real-time command and control). In particular, decentralized computers provide independent failure modes for reliability, multiple execution units for concurrency, and a flexible interconnection structure for extensibility.

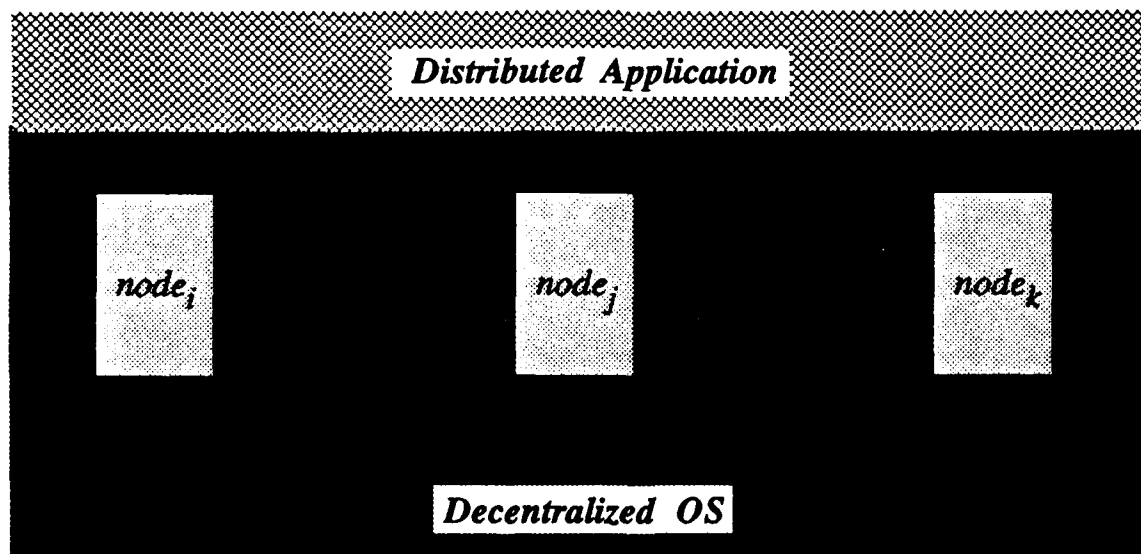


Figure 1: Example Decentralized System

It is often necessary or desirable that an application be distributed across multiple nodes in a decentralized computer system. A distributed application is far more integrated than individual applications which share resources or even interact. A distributed application is conceptually a singular entity created with a coherent, consistent perspective and set of objectives, not separate entities which each have their own independent

objectives without concern for, or even awareness of, others. The constituent computations of a distributed application may extend across physical node boundaries (for a variety of reasons), rather than each being confined to its own local address spaces and the resources on one node.

A decentralized operating system consists of replicated copies that constitute the native operating system on all the system's processing nodes. However, the system's resources are managed in a global fashion. This is done either by direct coordination among the various instances of the kernel, or locally by each kernel instance as consequences of the higher-level resource management decisions.

These system characteristics are quite different from those of more traditional multiprocessors, computer networks, and other systems with similar hardware structures. For example, in computer networks (such as shown in Figure 2) the resources at a particular processing node are managed by the operating system local to that node, and the collection of autonomous, local operating systems interact in limited ways (e.g., to support such application purposes as file sharing, mail, and remote login) [Lampson 81]. Furthermore, the emphasis in the work described here is on multi-computer systems which do not have shared primary memory, separating it from operating systems built on multiprocessor hardware [Jones 79, Ousterhout 80, Wulf 81].

It should be noted that the hardware structure of a decentralized computer system, as defined here, need not be different from that of typical local area networks (i.e., a collection of processing elements with private memory and local peripheral devices, interconnected by an interconnection network). It is usually the operating system software alone that separates decentralized computer systems from local area networks.

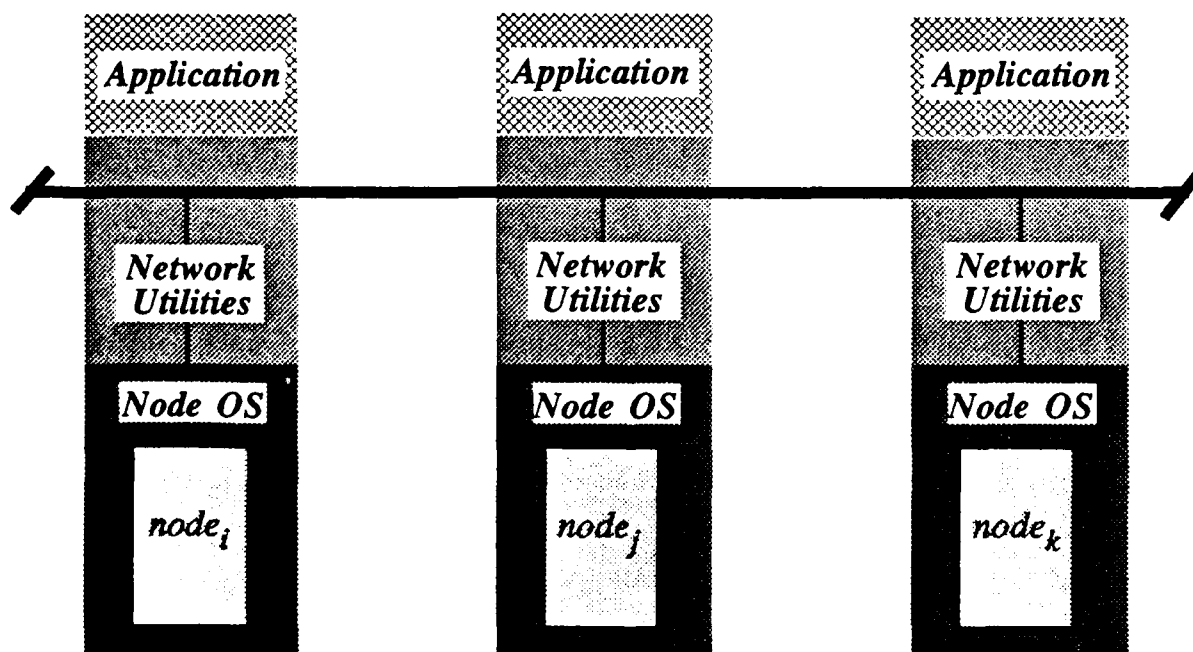


Figure 2: Example Distributed System

The class of distributed real-time command and control systems of interest in this research involves on the order of 10 to 100 processing nodes, physically dispersed across a distance on the order of 10 to 1000 meters.

2.2 Real-Time Command and Control

According to the definition used in this research, real-time computer systems control one or more physical processes whose states are sensed and altered by the computer system. The state of these processes changes independently as a result of the external environment, which is not completely under the control of the computer system. The principal distinction of real-time computing systems is that the physics of the application imposes time constraints (e.g., deadlines) on some of the computations. These constraints are not simply performance metrics, or an issue of programmer convenience, but are part of the correctness criteria for those computations.

The emphasis in real-time systems today is on a very specific subset of the general real-time application domain, and it is this area that is most typically assumed when the term "real-time" is used. The special case that the greatest amount of effort has been directed towards is the area of the lowest level sampled-data monitoring and control of physical processes—i.e., dataflow/pipelined signal processing or sensor/actuator feedback control. Furthermore, because these applications are inevitably considered to be rigidly periodic and deterministic, this popular view of real-time systems is not even an accurate perception (much less an optimal treatment) of the intended case.

There is a great deal more to real-time systems than is encompassed by the typical low-level view, however the larger real-time picture is based on the general notion that arbitrary computations may have constraints on their times of execution. In the most general real-time case, the application's time constraints may vary dynamically, the time constraints may not be predicted ahead of time, and not all of the computations that make up an application have time constraints associated with them at all times.

The class of real-time system considered in this research is known as *supervisory-level command and control*. Supervisory control is a middle-level function in the real-time application hierarchy—above the sampled data loop functions and below the human interface/management functions (see Figures 3 and 4). The area of real-time supervisory control is representative of the characteristics of the larger real-time application domain. Supervisory control systems do little direct polling of sensors and manipulation of actuators, nor do they provide extensive man/machine interfaces; rather, they interact with subsystems which provide those functions. In addition to having activities with critical time constraints, the behavior of real-time command and control systems is predominantly dynamic (i.e., aperiodic, asynchronous, and stochastic). Supervisory real-time command and control systems are found in plant (e.g., factory or refinery) automation, vehicle (e.g., airborne, aerospace, or shipboard) control, and surveillance (e.g., air-traffic control) systems.

Some tasks in real-time command and control systems are periodic and are bound to process activity rates, but most are aperiodic with stochastic parameters and are associated with external stimuli as well as the interactions of the system with low-level control subsystems [Boebert 78]. The real-time response requirements of a supervisory control system are closer to the millisecond than either the microsecond or second ranges.

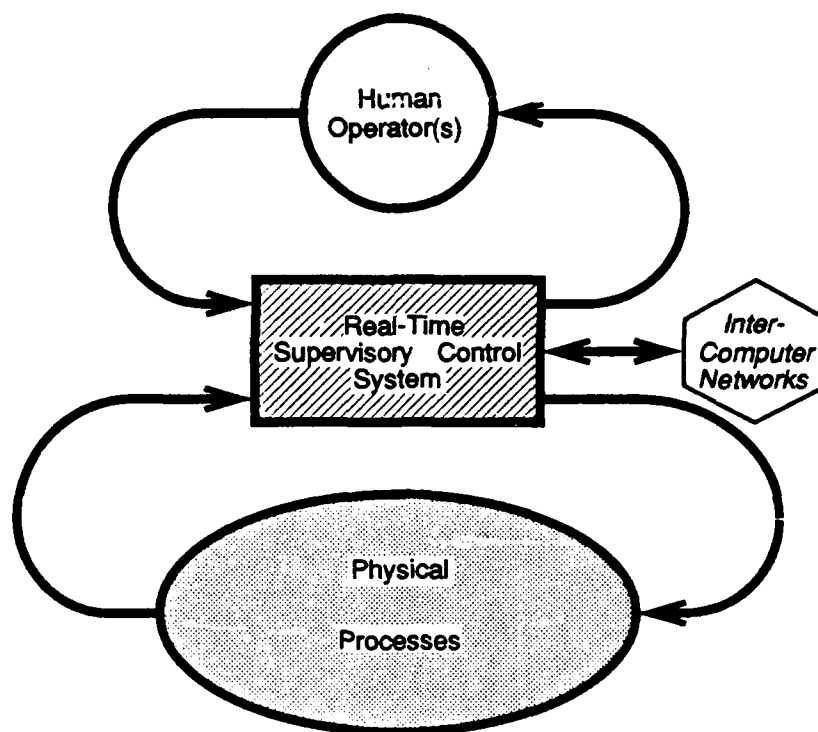


Figure 3: Real-Time Supervisory Control Context

Many command and control applications (and their I/O subsystems) are physically dispersed, and for this reason map well onto distributed computer systems. Furthermore, the distributed nature of these command and control systems suggest that they be implemented as distributed application programs. The concurrently executing computations within these distributed applications exhibit more cooperative behavior than those among other distributed applications. This is especially true of real-time command and control applications, where the concept of independent users does not exist as it does in typical timesharing and computer network systems [Thomas 78]. Rather, the whole system is dedicated to performing a particular mission that can be thought of as having a single user, comprised of the physical processes being controlled.

Level	Function	Operating System	Processor
4	Management Information System	VM (IBM)	3083 (IBM)
3	Plant Management System	VMS (DEC)	8800 (DEC)
2	Supervisory Control System	Alpha	Sun-3 (SMI)
1	Machine Control System	VRTX (H&R)	11/23 (DEC)
0	Sensors/Actuators	none	various

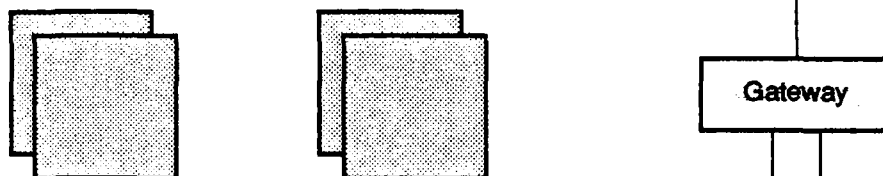
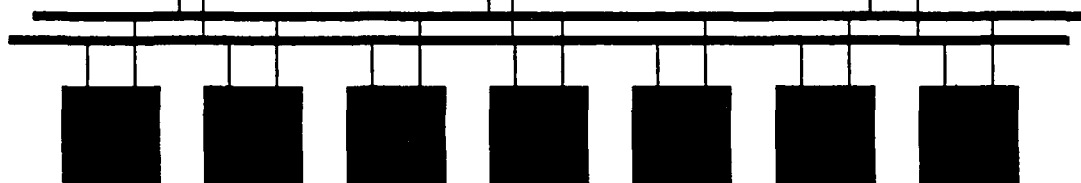
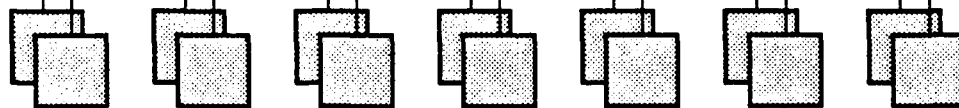
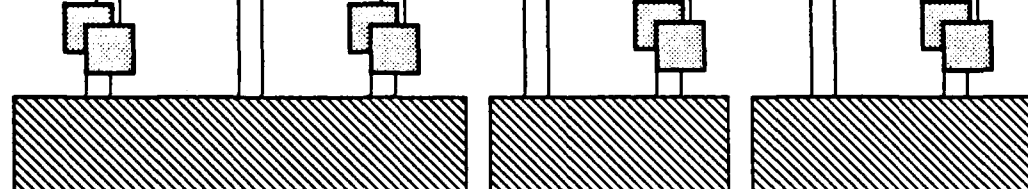
Level 4Level 3Level 2Level 1Level 0

Figure 4: Example Real-Time Systems Hierarchy

3 Special Requirements

A number of significant requirements accompany the distributed real-time command and control application domain. Some of these requirements are unique to this context, and others, while generally applicable to a wide range of systems, are especially important here. It is these application requirements that form the basis from which the Alpha operating system's programming abstractions and supporting facilities were derived. The requirements of this application domain can be grouped into four categories—i.e., features associated with the timely execution of programs, issues related to the physical distribution of the system, matters concerning the robustness of the system and application code, and the demands for adaptability at both the system and client levels.

3.1 Timeliness

In a real-time system, computations must be performed in accordance with the time constraints imposed by the physical processes being controlled. Furthermore, because of the dynamic nature of the external processes being controlled by a real-time system, some amount of the application's state information is time-dependent and its value degrades with time. For these reasons, the timeliness of the computational activities performed by a real-time system is an integral part of the definition of correct system behavior. In a real-time system it is not sufficient to ensure only that data generated as a result of a computation is correct and consistent, it must also be made available in a timely fashion for the result to be considered correct.

An operating system that is to support real-time applications must provide mechanisms which take these time-related issues into account and assist application programs in meeting their time constraints [Jensen 76b, Wirth 77]. There is a range of degrees to which a system can support the needs of real-time applications. A system could make it difficult to meet real-time demands (e.g., by trying to enforce a "fairness" policy in resolving contention for resources), it could provide mechanisms that help in the construction of real-time applications, or it could be structured so as to attempt to make real-time guarantees for its applications. Clearly, the term "real-time" does not define a simple dichotomy among operating systems, but rather there is a range of metrics with which any given operating system can be judged as supporting the needs of real-time applications. The more that a system does to enhance the ability of client programmers to create applications that meet their timeliness constraints, the greater claim the system has to being "real-time."

There are a number of standard requirements associated with all real-time systems; the least recognized, but most important of these is the notion of time-driven resource management. All of these requirements are discussed in the following subsections, along with some of the commonly held misconceptions about the general requirements for real-time operating systems.

3.1.1 Standard Requirements

Irrespective of their technical merit, there are some widely accepted requirements for real-time operating systems. Among these requirements are a number of features that are common features of modern operating systems that, because of advances in computer technology, have only recently become practical for use in real-time operating systems

(or are just being discovered by practitioners in the area of real-time systems—see [Glass 80]).

Real-time operating systems have been receiving a great deal of attention in recent times, however this attention has largely resulted in the rediscovery of features found in traditional, full-featured operating systems (e.g., dynamic multiprocessing, interprocess communications, address space protection, and high-level synchronization mechanisms). This is similar to how much of the work in microprocessors has resulted in the rediscovery of features found in traditional, full-featured mainframes (e.g., I/O channel processors and memory management).

In addition to those requirements that are common to modern operating systems in general, there are those which are primarily performance enhancements—e.g., fast context swaps, low interrupt latency, and high-bandwidth I/O. Other requirements affect only the implementation and are not directly visible to the client—e.g., allowing portions of the system to be paged, allowing preemption while executing within the kernel, providing for the streaming of data to/from contiguous regions on disk, and bounding execution times for system services. Some of these commonly held requirements have to do primarily with the system's implementation, but are visible to the client—e.g., the ability to pre-allocate system resources, and the ability to *lock* code/data into memory (i.e., prohibit the system's virtual memory subsystem from removing a region of primary memory). Yet other of these standard requirements are generally useful features that are visible to the client—e.g., the ability to access I/O devices directly from user space, the ability for clients to define (and load) custom device drivers, the ability to force writes to disk, and the ability to be asynchronously notified of specific I/O, exception, and application-specific events.

These standard requirements are largely taken for granted in the Alpha operating system research effort. These requirements are recognized as being important to the development of practical real-time systems, and the significant effort required to meet them is not underestimated. However, the focus of this research is on the other requirements defined in this chapter, because of the relatively large amount of experience and relatively small degree of complexity that the more common requirements exhibit with respect to the other requirements (e.g., distributed, reliable, time-driven resource management).

3.1.2 Time-Driven Resource Management

Fundamental to the definition of all operating systems is the management of system resources, and similarly the major factor in providing support for real-time applications has to do with how an operating system manages its resources. To support the needs of real-time applications, an operating system must manage resources in such a fashion as to attempt to ensure, to the greatest extent possible, that the timeliness constraints of its application programs are met. In addition to managing system resources in an efficient and consistent fashion (as all operating systems should), if the system manages its resources based directly on the actual time constraints of the application, this is known as *time-driven resource management*. According to this definition, the primary issue in the creation of real-time systems is the amount and degree of time-driven resource management that they perform.

In the extreme case, whenever a real-time operating system makes a decision as to how an instance of contention for some resource is to be resolved[†], it considers the impact of its decision on the timeliness of the computations that may be affected. Operating systems that resolve instances of system resource contention based on a metric other than timeliness (e.g., fairness or throughput), do not work to meet the application's time constraints and are not as qualified to be termed real-time systems.

An implication of time-driven resource management is that the operating system must be given (or be able to infer) the application's time constraints, and must be able to determine the impact of its resource management decisions on the application. In general, an operating system can only perform crude, brute-force types of resource management without (either explicit or inferred) guidance from the application. An adaptive, real-time operating system must receive (run-time or compile-time) inputs from its clients in order to resolve contention for system-managed resources in the desired fashion. Clearly, the more information that the operating system has, the better it can do in managing resources so the application's time constraints can be met. Similarly, the closer that an operating system's abstractions are to the programmer's abstractions, the less transformation must be performed (either by the programmer or the system) to map the application's timeliness requirements into the form needed by the system to manage its resources, and the easier it is for the operating system to determine the effects of its management decisions on the application.

3.1.3 Common Misconceptions

In addition to these requirements, there are some commonly held misconceptions concerning the requirements for real-time operating systems. Some of these beliefs are derived from valid requirements of specialized subsets of the general real-time application domain. However, most of these mistaken beliefs stem from primitive attempts at dealing with the difficult problems of real-time systems by attempting to cast the problem into a form that is more intellectually manageable, or that might be amenable to the application of some existing analytical tool. The most serious of these myths are that priority-based schedulers adequately support the needs of real-time applications, that real-time applications are predominately periodic in nature, and that poor system resource utilization can be used as an effective substitute for time-driven resource management.

3.1.3.1 Priority-Based Processor Scheduling

A common misconception about real-time systems is that all applications' timeliness requirements can be met with a simple priority scheduler. Firstly, it should be noted that scheduling is not the only resource management activity that an operating system performs. Because application software makes requests for resources primarily when a software unit has been dispatched, and because there is contention for processor cycles whenever more than one schedulable software entity in a system is ready to run, the management of processor cycles (i.e., scheduling) is an important and frequent resource management activity in any operating system.

[†]Note that the manner in which resources are managed is not significant if there is no contention for the resources—i.e., if there are enough resources to meet all of the demands.

The use of a priority scheduler does not go very far towards supporting the needs of real-time applications. In the form they are most often found, priority schedulers exhibit a number of problems. For example, when dealing with periodic activities, the notion of priority is derived (more or less directly) from application time constraints—e.g., static rate-monotonic priorities, and dynamic deadline priorities. However, the time constraints of aperiodic activities tend to be mapped in a much less methodological fashion into a narrow range of small integers intended to signify relative priority. This is because aperiodic activities are less amenable to analytical treatment, and integer priorities are thought to be more computationally tractable than actual time constraints.

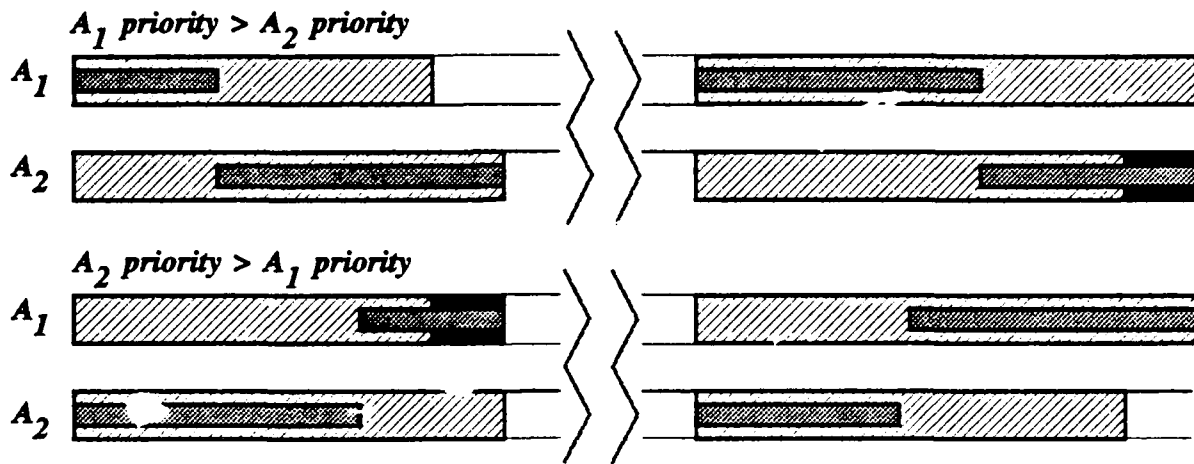
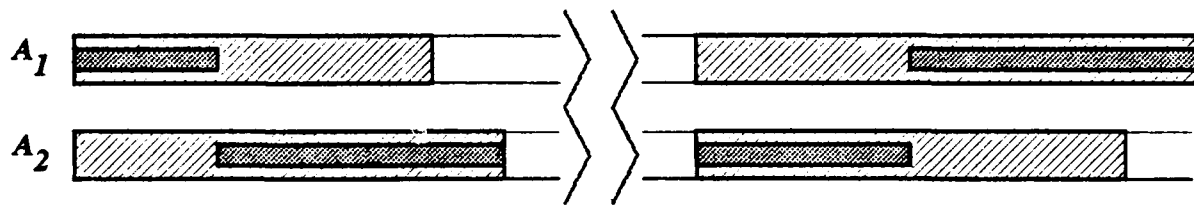
There are numerous deficiencies with the typical, *ad hoc*, uses of priorities. Included in the list of problems is the fact that much valuable information is lost in crude transformations from the time domain and into priorities, such as when a single metric is used to express two orthogonal characteristics of a computational activity—e.g., the timeliness requirements of a computation and the importance of its execution to the application. Furthermore, the range and resolution of the priority scale is often too limited to accommodate the number and gradation of characteristics of application activities. Fixed priorities also represent a premature binding of the dynamic information associated with computations that denies the system the opportunity to adapt to the many important changes possible at execution time—e.g., stochastic variations in load and resource contention, and faults, errors, and failures. The assignment of fixed priorities to computational units in a system requires a great deal of trial-and-error manipulation to achieve the desired effects. Furthermore, in some cases there does not exist a fixed priority assignment which allows all the activities to meet their time constraints, even though the system's resources might be under-utilized (see Figure 5).

Priorities which are dynamic overcome many of the adaptability and flexibility restrictions of static priorities, but even dynamic priorities (such as deadlines) suffer from serious disadvantages. Among the problems with dynamic priorities are: not being able to represent both the urgency and importance of computations, not being able to express both the *hard* and *soft* varieties time constraints, not having the power and expressiveness to depict a variety of different types of time constraints, and not being able to deal gracefully with overload conditions.

The real-time environment addressed by Alpha demands a type of scheduler that supports a dynamic expression of the timeliness constraints of each computation. Such a scheduler must also be able to determine how to allocate processor cycles to each of the application's schedulable software entities in such a way as to meet the application's time constraints whenever possible, and follow the system's overload handling policy when all of the application's time constraints cannot be met.

3.1.3.2 Deterministic Systems

Another myth is that real-time systems must behave in a highly predictable fashion in order for the system's performance "guarantees" to be met [Stankovik 88]. While many systems rely on the assumption that real-time applications are strongly periodic, the rigidly cyclic style in which typical (low-level) real-time systems are designed is rarely due to any inherent property of real-time applications. Instead, periodicity is only a particular technique which is believed to simplify the management of actual time constraints.

Static Priorities:Dynamic Priorities:




Execution Window —  Required Time —  Missed Window — 

Figure 5: Static versus Dynamic Priorities

This is despite the fact that many real-time applications include devices (and thus computations) which appear to exhibit periodic behaviors, such as: sensors which generate data on a physically cyclic basis; actuators which require recurring positional updates; and displays which need to be refreshed regularly. The classical periodic approach to expressing and handling such characteristics is to define a fixed iteration period for the cyclic, recurring, or regular activity, schedule all future iterations of that activity collectively in advance, and mutually synchronize (if possible) all periodic activities (e.g., into rate-groups) [Leinbaugh 80].

General real-time command and control systems and their applications cannot be constrained to behave in a highly deterministic fashion. Only certain, very stylized, computations (e.g., signal processing) exhibit such rigidly predictable behavior, and the underlying (possibly distributed) system cannot be constrained to behave in such an idealized fashion. Some periodic approaches are based on highly deterministic environmental assumptions—e.g., no exceptions occur during the system's execution, there are always sufficient resources to meet the application's needs, and activities always take the same

amount of time to complete. Operating systems that are based on such simplistic and unrealistic assumptions do not meet the needs of the greater space of real-time control problems. For larger and more demanding distributed real-time command and control systems, it is less reasonable to make such assumptions because the behavior of these systems is necessarily much more complex, dynamic, and stochastic, nor is it possible to constrain the system to behave in the desired fashion.

It is quite difficult to construct a system that is adaptive and yet able to meet demanding timeliness constraints, because adaptive distributed algorithms are difficult to create and tend to be costly in terms of performance. Nonetheless, such techniques must be developed to deal with the true needs of general real-time control applications.

3.1.3.3 Excess Assets

Another real-time myth is that an excess of system resources is sufficient to meet the demands of real-time applications. The belief that excess resources solves real-time application needs stems from the fact that if there is no contention for (or low-utilization of) system resources, then the resource management strategy employed by the operating system is not significant. It is true that the brute force approach of providing a sufficiently high degree of excess hardware assets, alone, would be capable (in principle) of meeting the computational needs of virtually any real-time applications. However, the complete suite of requirements for actual real-time systems cannot be met with such a simplistic approach. It is the demands of other system attributes, such as cost, size, weight, and power, that require system resource management schemes which do not depend on low resource utilization to meet application-level timeliness constraints. Also, practical experience shows that applications expand to fit the size of their systems, and that there are rarely enough resources to meet the basic needs of the application, much less the degree of excess assets needed to meet timeliness constraints in this highly non-cost-effective manner.

3.2 Distribution

One of the major functions of decentralized operating systems and network operating systems is to manage the inter-node communication resources for the clients. But in a distributed system, it is highly desirable that physical dispersal of the underlying hardware be made transparent at a low level in the system (i.e., the kernel). In this manner, both the system and the application programmers benefit from features such as physical-location-transparency, similar to how programmers benefit from a system-provided process abstraction. However, there are a number of cases (e.g., work assignment, redundancy management, specialized function location, or diagnostics) in which it is appropriate to provide clients of the kernel with information concerning physical location.

In a distributed system, system and application software is normally based on *thin-wire* [Metcalf 72], as opposed to shared memory, interconnection techniques. This suggests that performance, availability, and robustness would suffer if system mechanisms were to make use of centralized structures. Furthermore, the physical dispersal of the system hardware introduces variable and unknown communication delays that exacerbate the already difficult task of attempting to ensure deterministic behavior of the system. The kernel mechanisms for decentralized systems must therefore be adaptive and

deal explicitly with the effects of physical dispersal—i.e., inaccurate and incomplete information. Although it is below the kernel level and deals with a small number of simple, static resources, the ARPANET routing algorithm [McQuillan 80] provides an example of this type of behavior.

Integrating physically dispersed hardware into a logically unified system at the operating system level means that the clients of a decentralized operating system should not be required to be aware of, much less manage, the physical dispersion of hardware and software, nor the many complex consequences thereof. These issues should not be allowed to distract the clients from performing their application tasks. A decentralized operating system supports distributed applications without requiring users to perform the necessary distributed resource management. Such an operating system provides the client programmers with a coherent distributed programming model. It also provides the management of physically dispersed resources necessary to transparently integrate the system's processing nodes into an actual (not just apparent) single system. All of these functions should be performed by a decentralized operating system, without depending on any form of physically, or even logically, centralized control.

3.3 Robustness

In the context of this effort, robustness means the extent to which the application continues to function fully and correctly (including meeting the system's timeliness requirements) in the face of failures (which is meant to include *faults*, *errors*, and *failures* as defined in [Avizienis 78]).

3.3.1 Implications of Distributed Real-Time Control

The nature of the physical processes being controlled in real-time command and control systems is usually such that the safety consequences (in terms of personal or property damage) of not managing them properly can be quite severe. Thus, the robustness of a real-time control system is of utmost importance, often being more significant than the cost or performance of the system.

Furthermore, it is often the case that the controlling real-time system is embedded and cannot be maintained easily. Should a component fail, it can be quite difficult (if not impossible) to gain access to the system to perform the necessary repairs, and frequently repairs must be made without interrupting the normal functioning of the system. The *mission* times—i.e., the period of time when the system must continue to provide correct service, uninterrupted by either maintenance or repair activities—of real-time systems can range from hours to years, and there is no single adequate approach to dealing with the issues of robustness posed by each of these cases.

A decentralized operating system for the real-time control environment has special robustness requirements which are more elaborate than those for conventional network and distributed operating systems. For example, the function of providing physical location transparency of software modules is not complete without handling the automatic detection of, and supporting recovery from, node and interconnection failures. Furthermore, a decentralized operating system is itself a set of distributed programs and special-purpose distributed (both replicated and partitioned) databases. This fact has major implications on the construction of the system, and calls for the use within (not just

above) the operating system of robustness techniques inspired by traditional distributed databases, such as atomic transactions and the automatic management of replicated data.

To support the overall robustness goals of a distributed real-time command and control application, the system software must itself meet a certain level of robustness. In addition to this, the system must provide mechanisms that allow suitably reliable applications to be constructed. The operating system should not dictate a specific kind and degree of robustness, but rather it should allow its clients to choose what is desired for each individual set of circumstances, at a cost that is appropriate. Such a flexible, mechanism-based approach is conceptually more difficult to design, but supports increased system efficiency, in that the operating system's clients are free to make the appropriate cost/functionality tradeoffs on a case-by-case basis within their application programs.

3.3.2 Major Robustness Concepts

The distributed real-time command and control application domain calls for a set of operating system mechanisms that support the following robustness concepts: correctness of data and actions, availability of data and services, graceful degradation of function, and fault containment. While these concepts are useful in almost any system, they are critical in this application domain.

3.3.2.1 Correctness of Data and Actions

The correctness of the actions performed by an application is a function of time, sequencing, and completeness—i.e., the correctness of a set of actions is defined by the amount of time it takes each action to execute, the order in which actions execute, and whether, at the end of the set of actions, they were all successfully executed. The correctness of data is defined by its self- and mutual-consistency as well as its timeliness of visibility. It must be possible for the programmer to ensure that actions which manipulate data leave it in a consistent state, despite any failures that may occur in the course of the data manipulation.

To provide for the correctness of data and actions, an operating system must support some form of atomic transaction facility [Lampson 80, Moss 85]. Such a facility ensures the *atomicity*, *permanence*, and *serializability* attributes of atomic transactions are made to apply to computations that execute within the defined boundaries of atomic transactions.

3.3.2.2 Availability of Services and Data

The availability of services and data is defined to be the extent to which each service (and the data it encompasses) remains available to clients across system failures. Typically, a service that is statically and uniquely bound to a particular hardware functional unit becomes unavailable should that hardware unit fail. To increase the availability of services, failures that may result in service disruption must be eliminated, or the means must be provided for resuming the service elsewhere when a failure occurs. The dynamic nature of the physical processes being controlled implies that a great deal of information in a distributed real-time control system degrades with time. This dictates that the availability of information is at least as important as, if not more important than, its consistency. Under some circumstances in real-time applications there is little value in maintaining the consistency of a database if the information is unavailable for use when needed.

Similarly, when the correctness of the information in a database degrades over time, there may be little point in attempting to restore it to a meaningful consistent state following a period of unavailability.

For an operating system to provide for enhanced availability of services and data, some facility must be included to manage redundant copies of code and data—i.e., some type of replication scheme [Bayer 79] must be supported by the system.

3.3.2.3 Graceful Degradation of Function

Graceful degradation of function is defined to be the property of a system that permits it to continue providing the highest level of functionality possible as the demand for resources exceeds its currently available supply. In the context of this work, whenever contending requests for resources cannot all be met in an acceptable time, the contention should be resolved in favor of the functions that are most critical to the objectives of managing the physical processes being controlled. Information concerning the relative importances of individual application tasks can be provided only by the application programmer. To provide graceful degradation, the system could use this importance information to determine which tasks' needs will not be met, in order to sustain the highest and most useful level of functionality possible under a given overload condition.

In addition to providing a scheme for load-shedding under overload conditions, a real-time operating system should be based on hardware with separate, independent failure modes (e.g., a distributed system) in order to provide support for graceful degradation of functionality. In addition, the operating system must provide mechanisms for node failure detection and recovery (e.g., location-transparent communications and migration/reconfiguration mechanisms).

3.3.2.4 Fault Containment

Fault containment is defined to be a property that inhibits the propagation of errors among system components. If a failure occurs (or is induced) in a system or application component, the operating system mechanisms should limit (or assist its clients in limiting) the extent to which the failed component can adversely affect the behavior of others. For example, the operating system should not permit a failed software component to arbitrarily modify the state of other components, either intentionally or by accident. (This point of view is in direct opposition to the current trend toward "light-weight processes.") Furthermore, for this property to hold, mechanisms must ensure that a failed software component cannot consume resources in an unconstrained manner and thereby interfere with other, presumably good, software components.

While some degree of fault containment can be provided at compile-time, in many cases the only effective way of enforcing the containment of faults is by way of system-provided hardware mechanisms (e.g., a memory management unit). In order to achieve the benefits of fault containment, the price of enforcement must be paid. While it is clear that most systems can be made to run faster if they do not provide defensive protection measures, it is not as clear that the robustness increase is not worth the price paid in terms of performance (especially in systems where robustness is a critically important attribute).

3.3.3 Optimizing for Exception Cases

An additional robustness characteristic of real-time systems is that the correct and timely execution of a real-time command and control application is typically more important under exception conditions than in normal cases. Also, the nature of exceptions (such as hardware failures due to physical damage) in these systems is such that they tend to be clustered in both time and space—as opposed to being randomly (e.g., normally) distributed, as is commonly presumed in other contexts. These characteristics have a significant impact on the nature of the fault tolerance and recovery techniques used in systems, and are contrary to the premises underlying almost all non-real-time computing system approaches (e.g., RISC-style operating system philosophies and the “end-to-end” argument) [DRC 86]. The designer should make an effort to perform reasonable engineering tradeoffs in the system design to ensure that the proper emphasis is placed on the exception conditions and not have the system be optimized exclusively for the expected, non-exceptional cases.

Figure 6 provides an simplified illustration of the types of tradeoffs typically made in non-real-time systems (6a), and the objective that real-time systems should aspire to (6b). It should be noted that a higher degree of steady-state overhead may be tolerated by real-time systems in order to reduce the magnitude and variance of performance resulting from exceptions.

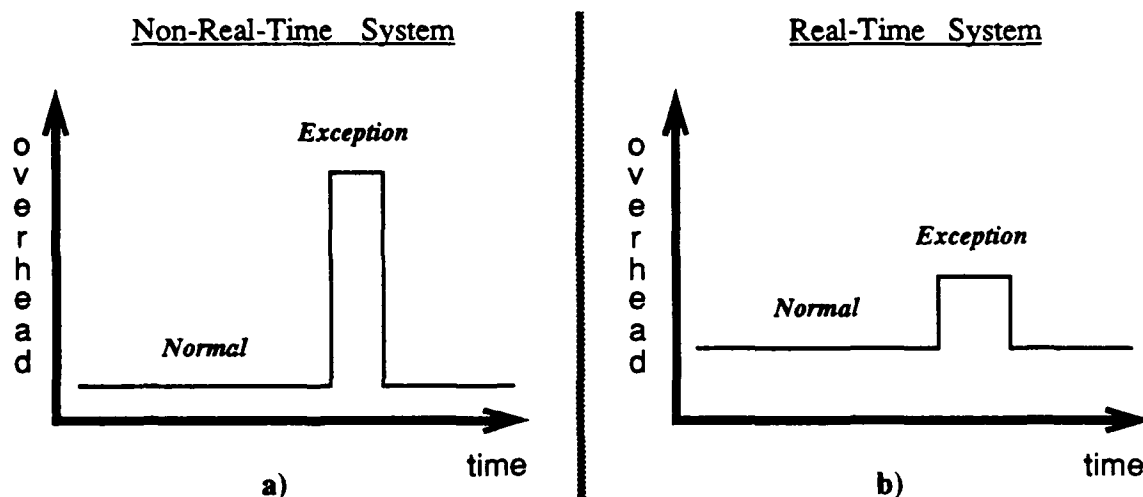


Figure 6: Optimizing for Exception Cases

3.4 Adaptability

In real-time command and control systems, modifications, technology upgrades, testing, maintenance, and other life-cycle items invariably make up the most significant portions of a system's cost [Savitzky 85]. Typically, the software-related costs predominate because the requirements are poorly understood at system design time and continue to evolve throughout, not just the design and implementation phase of the system, but even during the system's lifetime (which may be a decade or more) [Boehm 81, Lehman 85]. The principle reasons for this are that: the application is extremely complex and is not

(perhaps cannot be) well understood; the application environment (e.g., the product or threat) varies according to economics, geopolitics, doctrine, etc.; computer technology changes rapidly, making some desirable system attributes possible and others more cost effective, but also making older technology unaffordable or inaccessible; and the computing system is frequently viewed as the ultimate recourse for accommodating system changes (because modifications are always just "a simple matter of programming").

This implies the need for system software to provide a programming model that supports such desirable software engineering attributes as adaptability and maintainability. Because these attributes are more frequently associated with languages than operating systems, an operating system for real-time command and control might facilitate adaptability by taking into account the run-time packages of languages that address these attributes of adaptability.

In the realm of real-time command and control, system sizes range from quite small (e.g., an individual vehicle), to very large (e.g., an entire plant). Furthermore, in a decentralized system, processing nodes may be added and removed (either statically off-line, or dynamically due to the run-time failure and recovery of hardware). The operating system must itself be able to function effectively across a wide range of application and system sizes to take advantage of the opportunities for extensibility offered by the inherently modular hardware architecture of a distributed system, and to provide the robustness available from reconfiguration.

The need for the properties of modularity and extensibility are not unique to decentralized computer systems. Centralized systems may provide the necessary level of performance for a given application and may incorporate system software that is quite modular and extensible; however, centralized uniprocessor and multiprocessor hardware pose quite severe limitations in these respects. Decentralized systems offer a wider range of cost/performance choices than is usually available from a family of centralized processors or shared-memory multiprocessors. Overall, the physical properties of decentralized systems offer the potential for greater robustness, extensibility and adaptability, both within the system as well as in the applications [Franta 81].

Modularity and extensibility (even more so than robustness) are considerably more difficult to quantify and measure than other system attributes, such as performance. This may account for the fact that concern for performance (usually in the form of throughput) often dominates other considerations, fostering the misconception that these other system attributes are of lesser importance. This is unfortunate since system performance increases are derived automatically from advances in semiconductor technology, while increases in system robustness, adaptability, and extensibility result almost exclusively from thoughtful effort by system designers.

4 Current Practice

Operating systems in existence today (even those claimed to be "real-time") do not adequately address the requirements posed by the type of distributed real-time command and control applications described here. While some systems deal with certain aspects of the overall problem, it is safe to say that none meets even a significant number (much less all) of them. In the following sections, current operating system practice is examined with respect to each of the previously defined major requirements areas for real-time command and control applications.

4.1 Timeliness

It is not reasonable to characterize operating systems simply as being real-time or not, but rather there is a spectrum, into which all systems fit, that defines the degree to which the system can meet the demands of various real-time applications. The aspects of operating systems that increase their suitability for real-time applications are not manifest merely in the inclusion or exclusion of specific functions, nor does the inclusion or exclusion of any functionality, in itself, make an operating system unsuitable for real-time use.

Today there is range of meanings for the term "real-time operating system," as evidenced by the widely disparate types of systems that are referred to as being real-time. The applicability of this term to a particular piece of system software can best be judged by examining two fundamental aspects of the system—namely, the extent to which it is an operating system (i.e., the kind and degree of resource management that it performed), and the extent to which it is real-time (i.e., the kind and degree of support for meeting users' time constraints that is provided).

Today's real-time "operating systems" vary widely in the kind and degree of resources they manage, programming models they support, and the degree to which undesirable artifacts of the underlying hardware are obscured. For example, the types of system software that is sometimes referred to as "real-time operating systems" range from simple rate-monotonic dispatchers (e.g., for avionics [GD 80]), to minimum functionality embedded-system executives (e.g., VRTX [Ready 86]), to communication-based non-real-time kernels (e.g., V [Cheriton 84]), to full-functionality non-real-time operating systems (e.g., UNIX).

The "real-time" operating systems of today also vary in the extent to which they support the needs of real-time programming. For example, things claimed to be real-time operating systems range from non-real-time systems (e.g., UNIX and V), to non-real-time systems with "real-time" extensions (e.g., RTU [Henize 86]), to non-real-time systems with extensions plus kernel enhancements such as preemptability (e.g., MACH [Accetta 86]), to priority-based "faster-is-better" systems (e.g., VRTX), to static, periodic-based dispatching systems (e.g., SubACS [Wallis 84]), to highly general and adaptive real-time operating systems (e.g., Alpha [Northcutt 87]).

4.1.1 Minimalist Systems

To an overwhelmingly large extent, the system software that is referred to as "real-time operating systems" today is aimed at solving problems at the low end of the spectrum of real-time applications. These tend to be minimal-functionality systems, based on

the assumption of periodic application behavior, they frequently use static priorities to resolve contention for resources, and make use of low resource utilization in order to obtain hollow performance "guarantees."

This type of real-time "operating system" is merely vestigial compared with the normal meaning of the term, and are often more accurately termed "executives." Some such minimal systems (like those found in current avionics computer systems) are little more than rate-group dispatchers, and most provide only a library of rudimentary resource management routines. For the most part, they provide minimal functionality—preferring instead to pass the time, space, and intellectual complexity burdens of system resource management on to the application programmer.

These small executives strive to avoid doing anything that would make it difficult for applications to meet their time constraints, and try to provide service to the clients in a predictable fashion (primarily with respect to time); they incorporate little more than priority interrupt handling and context swapping facilities in an attempt to facilitate real-time responsiveness. The higher levels of functionality are typically left to the users in the belief that they can implement exactly what they need as a part of the application software, thus reducing execution time and memory consumption (however, this strategy frequently backfires).

Minimalist real-time operating systems arose from a historical context when processor cycles and memory space consumed a far greater proportion of the system's cost/size/weight/power than they do now. Such simple system software approaches may be adequate when the application software is correspondingly elementary, but the evolutionary trend (even in the area of avionics) is toward rapidly increasing system functionality and complexity. The effects of this trend can result in recurring, non-standard, lower performance attempts by users to perform system (as well as application) resource management—an especially expensive mistake with distributed systems.

4.1.2 Priority-Based Scheduling

The conventional approach to dealing with timeliness in current real-time systems is based almost exclusively on one or more of the following functions: static, priority-based processor scheduling, priority interrupts (typically statically assigned); fast interrupt handling and context swaps; fixed-priority (e.g., rate-group) scheduling, performed based on *a priori*, static pre-allocation of resources; and extensive configuration analysis, simulation, testing, and tuning. Notably missing from this list is anything that assists in the time-driven management of resources. Of all of the techniques used to make a system be real-time, by far the most common one is priority scheduling.

Instead of dealing directly with the application's time constraints, current real-time computing systems attempt to map them into artifacts which they (usually implicitly) imagine to be simpler, but often there are wide-spread detrimental consequences whose cause is not recognized. The principle forms that these artifacts take are the use of static priorities and the dependence on strongly periodic applications behavior. Such systems provide only a priority mechanism for managing processor cycles; the difficult task of making the priority assignments is left to the client, and typically requires a great deal of tuning. It is also common to pre-define multiple sets of fixed priority assignments and overlay them to meet certain anticipated combinations of changes (or modes) in the system.

Attempting to devise appropriate priority assignments for an application's computations is often an extremely complex and time-consuming effort that is relegated to highly-experienced specialists, and is (almost necessarily) dealt with in an *ad hoc* fashion. Furthermore, when changes are made to components of an application constructed using static priorities, it is frequently the case that the entire process of priority assignment and testing must be repeated. In addition to these deficiencies, it has been shown that, in general, fixed priority assignments are incapable of meeting activity deadlines, even when the computing capacity is far greater than the needs of the activities [Liu 73].

Provided that certain (often rather unrealistic) assumptions are met, some static priority-based scheduling techniques can be used to meet the time constraints of some types of real-time applications. For example, the *rate-monotonic* technique is known to be an optimal static algorithm with respect to meeting periodic deadlines [Liu 73]. However, only trivial systems manage to achieve a successful balance between priority responsiveness and resource utilization.

The solution to many, but not all, of these deficiencies is to use a dynamic priority scheme. However, despite the fact that dynamic priority assignments can significantly out-perform static ones, such techniques are rarely used in actual real-time systems.

4.1.3 Low Utilization

Some real-time systems attempt to guarantee that all of an application's timeliness requirements are always met. While it would be ideal for a real-time operating system to provide such timeliness guarantees, current approaches to making them introduce significant programming constraints and usually require that the system conform to certain unrealistically over-simplified assumptions concerning its behavior (e.g., as in [Leinbaugh 80]). Most efforts in this area focus on providing high resource utilization for low-level (i.e., closed-loop, sampled data) control applications where tasks are deterministically periodic and have no value to the system if their deadlines cannot be met (i.e., tasks have hard deadlines). The appeal of such approaches is their analytical tractability, but they are not suitable for more general real-time command and control contexts, which are characterized by predominately aperiodic tasks and are less amenable to such rigid and stylized treatment. In general, it is currently no more practical to make absolute timeliness guarantees for general, unconstrained real-time systems than it is practical to prove the correctness of large, complicated programming systems. Recently, some of the vocal academic proponents of real-time guarantees have begun recognizing the futility of their position and now speak of "conditional guarantees" [Stankovik 88].

4.2 Distribution

Despite the fact that physically dispersed systems are becoming more wide-spread, virtually none of them attempts to be globally unified in the previously defined manner. Most distributed systems are simply networks, or at most federated (dedicated-mission) systems, functioning as loosely associated collections of autonomous computing nodes with the ability to interact with each other [Thomas 78, Lampson 81]. In typical distributed systems, each node manages its local resources independently, and the client's interface to the system is usually non-location-transparent and non-uniform.

Ideally, the programming of distributed applications should be focused on the intrinsic issues of logical structure and algorithms, but distributed programming does depend on

distributed resource management—i.e., the management of physically dispersed resources so as to maintain application correctness, consistency, and performance in the context of such complexities as: asynchronous (real) concurrency; variable, unknown communications delays; and multiple independent failure modes. Distributed resource management in this sense should be the responsibility of the system so that the users can devote their efforts to the application programming. However, most conventional distributed operating systems manage system resources locally (i.e., per-node) and only a few resources (e.g., inter-node communications) are managed globally across the whole system. Therefore, almost all of the distributed resource management for distributed applications must be provided by the users, at substantial recurring development costs, and at similarly high, recurring, performance penalties.

In most distributed systems today, the programmer of distributed applications must explicitly be aware of, and deal with, the effects of physical distribution (e.g., the physical locations of software modules, the number of nodes in the system, and current state of the each node). This is because most conventional “distributed” operating systems are simply centralized, local operating systems with additional standardized interfaces and protocols intended for certain forms of resource sharing among separate applications executing in a network of autonomous nodes—e.g., shared file access or remote procedure calls to servers. Usually this sharing of resources is not uniform and transparent to the applications; the system’s clients must access remote resources in a manner much different from how local resources are accessed. Furthermore, the interfaces and protocols which are added to standard operating systems to make them “distributed” are typically implemented as utilities at the higher layers of the operating system. While this design decision serves to minimize the impact of distribution on local operating systems, it provides support for distribution at a higher cost than “native” implementations.

Typical centralized operating systems are extended to be “distributed” by adding a remote file system, or remote procedure calls between client and server processes. The resulting system provides a programming model that is inadequate for distributed applications in that it does not address some of the most fundamental issues in distributed programming, leaving them instead to be dealt with at the application level. First, such distributed systems typically do not provide a strong correspondence between the application’s viewpoint of a computation which spans multiple nodes and the system’s viewpoint of separate computations on interconnected nodes. To the underlying system, the components that make up the logical computations in a distributed application program are disjoint entities; the system does not see any relationship among them and therefore must manage system resources without the benefit of knowing about the relationships that bind them into a common logical computational activity. In addition, it is incumbent upon the application programmer to transform the structure of his logical computations into components that are supported by the system. The programmer’s logical view of a computation is the only thing that holds these independent components together—they all look the same to the system that is managing resources for them.

In addition, the most common type of distributed system does not provide the means for propagating a logical computation’s attributes (e.g., real-time constraints and reliability needs) along with it, as it progresses through the various components it makes use of in the course of its execution. This means that the system does not maintain the continuity

of the acquired attributes of a logical computational stream as it passes from one module to the next (regardless of whether it involves crossing node boundaries). This makes it impossible for the system to maintain the continuity of the information that is associated with a computation, as opposed to the various components that go into making up each computation in an application. Because the system does not maintain this continuity, it cannot manage resources effectively and there is only a limited degree to which an application can make up for this deficiency.

Most distributed systems in existence today tend to provide limited, stylized support for concurrency and synchronization of distributed computations. These systems tend not to provide support for either concurrency within a (multiprocessor) node, or concurrency among the nodes in a system. But rather, they provide the type of process management and interprocess communication facilities that were developed for uniprocessor or network systems. Additionally, the forms of synchronization dictated by these systems tend to limit the amount of concurrency that can be obtained within a programming module. This is typically done because the systems that these structures were developed for did not have any actual concurrency to exploit, and so restricting concurrency has the beneficial effect of simplifying things for the programmer. If the system restricts concurrency within a programming module to where only one point of control can be active in it at a time, the programmer need not be concerned about synchronization, and can write the usual form of non-reentrant, straight-line, serial code.

4.3 Robustness

A common technique for attempting to provide robustness in real-time control systems is through the extensive use of excess assets—i.e., the system is provided with far more resources than necessary to meet the application's (steady-state) computational demands in order to ensure that the system's objectives can continue to be met in the face of failures. These systems rely on a very low level of average system resource utilization in order to achieve their robustness goals.

Most extant systems that address the robustness requirements described in the previous sections tend to be special-purpose database systems that use low-concurrency, high-cost consistency maintenance schemes (e.g., atomic transactions), or equally costly redundancy management techniques (e.g., replication). The use of these techniques has not been directed towards the area of distributed real-time command and control, and so little effort has been expended in trying to develop robustness techniques that also meet the timeliness demands of real-time systems. Frequently (regardless of the specific techniques used) a system's robustness facilities impose substantial constraints on the system and application programmers, and furthermore, most of the systems that have been constructed to explore techniques for reliability emphasize only one aspect of system reliability. The reliability facilities in current systems tend to impose fixed costs for their use, irrespective of whether the client currently needs the system's robustness services. Furthermore, the reliability facilities currently being developed are not geared to meet the special needs of the real-time command and control area, and are therefore not well integrated with the other facilities required by this application domain.

Regardless of the particular mechanisms used, a common technique for ensuring that a real-time application exhibits the proper degree of robustness is exhaustive testing.

Despite the fact that it is widely known that testing only detects the presence of bugs and does not ensure the absence of bugs, testing is used to obtain a greater level of confidence in systems (and configurations). As it is currently practiced, the exhaustive testing of real-time systems is a dead-end technology, that is already being pushed to its limits, and will not hold up to the demands of future systems. As real-time command and control systems have become more complex, it has become impossible to exhaustively test them. With larger numbers of processors carrying out larger numbers of tasks, the combinatorial explosion of paths does not allow simple exhaustive tests to be written and applied with any degree of confidence that a high degree of coverage can be obtained. As systems become more dynamic, it is no longer possible to anticipate, with a small number of load modules, all of the system configurations that may result from subsystem failures. The testing issue has long acted as a force to keep real-time systems small, simple, and deterministic. What is called for is the advancement of the state-of-the-art in testing (as well as design-for-testability) technology to cope with the advancing sophistication of the systems instead of using testability as an excuse for constraining the systems.

4.4 Adaptability

While certain existing systems have emphasized modularity and extensibility in their designs, they have not been systems of the type that is of interest in this research. Although some real-time control systems exhibit a significant degree of architectural modularity (via the use of processing nodes interconnected by high-performance communication networks [Jensen 78b]), and others provide adaptability at the programming language level (via a high-level language such as Ada), there are few examples that have explicitly attempted to be usefully modular in their system-level design. Likewise, there are operating systems that provide a high degree of modularity and extensibility at their interfaces and within their structure, however few of these have been distributed systems and fewer still have been for real-time command and control.

4.4.1 Dynamic System Behavior

Currently, most real-time systems attempt to make the behavior of the system and applications as static and deterministic as possible, and attempt to validate the correctness of the system through exhaustive testing [Quirk 85]. A common characteristic of this type of system is the high cost associated with the addition or modification of system functionality, and the inability to cope with unanticipated behavior [Parnas 77, Glass 80].

In real-time control environments the value of flexibility has not always been properly appreciated because the computing system is usually embedded. The pervasive philosophy is that a static system (that is analyzed, configured, tested, tuned, and validated *a priori* for as many contingencies as possible) meets all of the needs of real-time applications, is more dependable, and maximizes performance with minimum assets. However, factors now mitigate in favor of greater computer system flexibility than has historically been available in this environment. Many applications are becoming so sophisticated that the common static approach is clearly infeasible, and computer hardware technology advances have diminished the size, weight, and power per unit of performance, as well as the percentage of system costs represented by computer hardware assets.

4.4.2 Dynamic Time Constraints

The greatest impediment to adaptability in real-time systems results from their approach to managing time constraints—most take an approach that depends on periodic behavior of the system to make the problem more manageable, or amenable to analytical treatment. As outlined in the requirements section of this report, real-time applications themselves are not necessarily so strongly periodic. While there are many cyclic activities in real-time systems, the actual time constraints for many types of repetitive-sample activities can be expected to have a significant degree of variability (e.g., due to the mechanical tolerances of the physical environment). Even if the application does impose strictly periodic requirements, there are aspects of the application's implementation and the underlying system that introduce variance into the system. This periodic-based approach is not required by real-time applications, it does not adequately address all of the real-time application problems, and it restricts the flexibility of systems in a number of different ways.

In particular, the periodic approach does not deal with the variability that accompanies real systems. Many forms of variability are inherent in any (non-trivial) real-time system—e.g., stochastic fluctuations in load and resource contention, mechanical tolerances in sensors and actuators, and faults, errors, and failures of system components. A periodic-based approach attempts to, in an *a priori* fashion: anticipate certain variabilities and define away the majority of those that remain; prohibit architectural optimizations that introduce nondeterminism (such as caches, virtual memory, and direct memory access channels); design, and provide resources, for the “worst case”; dedicate and pre-allocate all system (and application) resources; test and tune the system until all cases tested “work”; and “guarantee” system behavior (i.e., response times). This results in rigid, brittle structures which must be invested with massive quantities of resources, and these assumptions must not be violated or else the system will shatter catastrophically under the pressure of variabilities (e.g., ancient bridges and cathedrals which were rigid and massive).

Furthermore, a periodic-based approach does not allow the run-time evolution of time constraints. In periodic-based systems, once the period of a repetitive activity has been *a priori* established, tested, tuned, and validated, it must remain fixed despite any run-time changes that may occur in the application environment. There are many cases where this is a serious limitation. For example, where the interval between successive repetitions may evolve dynamically throughout the course of a periodic activity's execution—sampled-data monitoring/positioning can occur with increasing frequency as a robotic manipulator approaches an object, a pair of controlled vehicles approach each other, or a variable gets closer to its limit.

Additionally, the periodic-based approach to real-time system construction does not deal properly with aperiodic events. In many important real-time environments (such as real-time command and control) time-critical aperiodic activities are a major factor, and are the predominant form of time constraint. Conventional periodic-based real-time systems attempt to force-fit aperiodic activities into their periodic mold—e.g., by providing excess capacity reserved for aperiodic activities in a background frame, or by providing a periodic server which multiplexes aperiodic activities. But none of these schemes is capable of effectively scheduling activity mixes in which a significant percentage are ape-

riodic activities having time constraints (such as deadlines), or efficiently utilizing resources.

Both periodic and aperiodic activities fit naturally in an aperiodic-based approach, thus it is the more general. An aperiodic-based approach provides a more general solution to the problem of time-driven resource management that does not suffer from the periodic-based approach's lack of flexibility. With an aperiodic-based approach, periodic activities can be expressed and handled more generally and directly. For example, sensors can be read, actuators updated, and displays refreshed at specific time intervals (within a specified tolerance). Each time interval occurrence can be managed individually (like loop unrolling) as though it were one more of an unrelated series aperiodic events. Such an aperiodic-based approach meets the demands of aperiodic events as well as exhibiting significant advantages in dealing with the elasticity and evolutionary needs of periodic activities. While periodic-based approaches are the exclusive focus of both practice and theory today, the Alpha operating system departs from this historical trend and represents the first step in the development of the alternate (more promising), aperiodic-based technology.

5 Technical Approach

Alpha was designed to meet requirements defined in the previous chapter, without suffering from the shortcomings of existing systems. This research was directed toward the synthesis of new concepts to meet the particular needs of the previously defined application domain. This required a carefully integrated software/hardware approach, along with the exploration of the tradeoffs required to implement those concepts effectively. One of the results of this research was the creation of a set of programming abstractions that are intrinsically well suited to modular, reliable, decentralized operating systems, and the design and implementation of a set of kernel-level mechanisms in support of them. Alpha is not a monocentric system, that features one particular concept over all others, nor is it a collection of facilities grafted onto an existing operating system. All of the system's features were designed based on the requirements for the application domain and implemented in concert with each other, resulting in a well-integrated solution within a meaningful framework.

Alpha was designed to be executed on loosely-coupled architectures, with either uniprocessor or multiprocessor nodes. Additional constraints placed on the implementation of Alpha include the requirement that the system be able to run on standard, off-the-shelf processors (i.e., the system cannot be dependent on specialized hardware support to make its implementation practical), and the requirement that the system's (native) programming model be similar in concept and implementation to traditional models (i.e., the system cannot force its programmer's to learn and adopt a radically different approach to programming).

Alpha's kernel provides a small, simple set of mechanisms, out of which the higher levels of the system, and the applications, are constructed. These mechanisms permit the exploration of a wide range of system policies, and provide great flexibility for implementing other programming models.

It should be noted here that the Alpha project is directed towards the development of operating systems technology and, as such, the emphasis has been on providing a set of system mechanisms to meet the requirements defined earlier. The intent is that multiple, differing, programming interfaces can be supported by Alpha, ranging from elaborate distributed real-time programming environments, to (more or less) standard object-oriented programming languages, to some standard programming language with direct access to the native system interface. The implementation work on Alpha has been proceeding bottom-up, with the kernel being complete (and somewhat stable) at present, and there is currently no object-oriented programming language (standard or otherwise) implemented on top of the kernel. Instead, the programming experience with Alpha to date has been performed directly on the kernel interface, using a pre-processor to provide the programmer with a set of simple language constructs for programming with the native abstractions provided by the kernel of Alpha [Shipman 88]. Throughout the remainder of this report, the programming abstractions and features of Alpha are discussed from the perspective of programming the native, kernel interface of the Alpha operating system (despite the fact that this is not intended to represent the ideal, or only, programming interface provided by Alpha).

The system's basic programming abstractions were designed explicitly to support the requirements of timeliness, distribution, robustness, and adaptability. In the following sections, the basic abstractions provided by Alpha are defined, the means by which clients of the kernel specify time constraints (i.e., the notion of time-value functions) is described, and the structure of the system's software and hardware is briefly outlined.

5.1 Basic Abstractions

The Alpha kernel interface presents its clients with a set of simple and uniform programming abstractions from which modular, reliable, and distributed real-time control applications may be constructed. The purpose of a kernel is to provide fundamental abstractions and mechanisms that support a range of different system interfaces (i.e., operating systems and languages, similar to [Habermann 76]), which is not the same as a trivial operating system or an executive. The Alpha mechanisms are carefully and deliberately devoid of policy decisions, and are meant to support the exploration of a wide range of decentralized operating system policies. The interface provided by the kernel is not (necessarily) intended to be the same interface presented by the operating system to an application programmer.

Alpha's kernel is based on a small set of basic mechanisms, in the same spirit as those in Accent [Rashid 81]. The Accent kernel is based on the process and interprocess communication abstractions, while the Alpha kernel is based on the abstractions of objects, the invocation of operations on objects, and threads. As in Accent, where system calls are performed by sending messages to processes, all kernel services in Alpha are provided by the invocation of operations on objects.

The abstractions provided by the kernel of Alpha are based on a combination of the principles of object-orientation [Bayer 79], atomic transactions [Bayer 79], replication [Randell 78], and decentralized real-time control [Jensen 76a].

The general programming paradigm supported by the kernel of the Alpha operating system is known as *object-oriented programming* [Goldberg 83, Cox 86], and the primary abstractions supported by the kernel are: *objects*, *invocations*, and *threads*. In Alpha, *objects* adhere to the common definition of abstract data types and interact with other objects via the *invocation* of operations on them. *Threads* are defined to be the manifestations of control activity (i.e., the units of concurrent computation and scheduling) within the kernel.

5.1.1 Objects

A wide range of benefits are claimed for the object model of programming, including increased modularity, the separation of specification from implementation, and the increased reusability of software components. These claims are accepted as true in the Alpha project, and a general discussion of the merits of object-oriented programming can be found in [Cox 86].

In addition to the attributes of modularity, information-hiding, maintainability, etc. normally associated with an object-oriented programming paradigm, the programming model described here is especially well suited for the support of decentralized, high-concurrency implementations of the major robustness techniques supported by Alpha (i.e., atomic transactions and replication).

At the highest level of abstraction, objects in the Alpha kernel are equivalent to abstract data types—i.e., some encapsulated data along with the code for a set of operations with which the data is manipulated. Objects are written by the programmer as individual modules, composed of the object's data and operations that define its interface (similar to Ada packages [Ada 83]). An example object is shown, schematically, in Figure 7, illustrating the data encapsulated by the object and the operations that make up the object's interface.

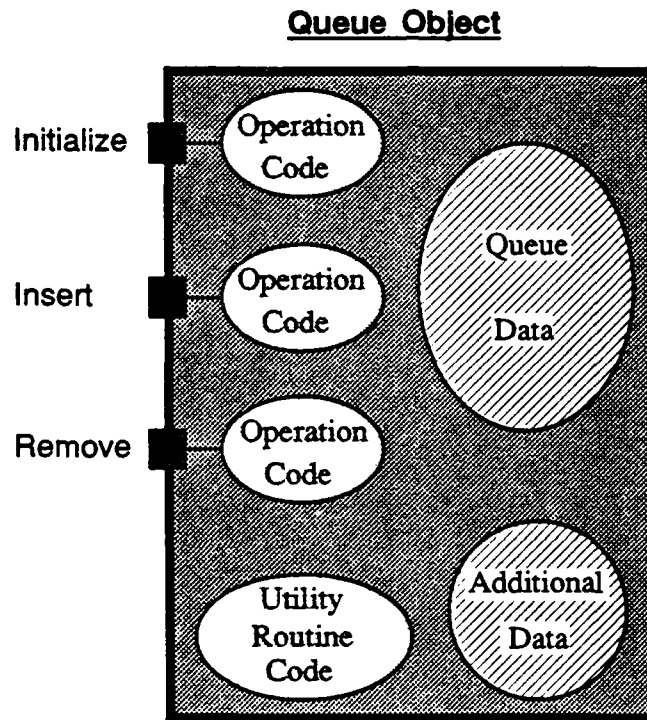


Figure 7: Example Object

Object *types* are static definitions of the code and the initialized data that go into making up an object. Object types are passive entities, defined by the programmers and maintained within the system object store. Object *instances* are the executable, run-time manifestations of objects that are dynamically instantiated from given object types, and may likewise be dynamically deleted from the system. In the following discussion (except where explicitly noted otherwise), the term *object* refers to an instance of a specific object type.

The basic abstractions of Alpha were developed to meet the system's requirements and therefore the definition of objects in Alpha differs from the more commonly accepted definitions. For this reason, the names of the Alpha programming entities were explicitly chosen to carry as little as possible of the semantic baggage associated existing object-oriented programming terminology (e.g., messages, methods, and classes).

In Alpha, objects are simple, passive entities, the main aspects of which are the rigid encapsulation of state information and clear definition of an interface to the encapsulated information by a set of operations. To enforce the encapsulation of information, each

object in Alpha exists in a separate, hardware enforced address space, and to enforce the integrity of the interface projected by an object, the kernel ensures that execution within an object can begin only at entry points specified by the object's exported operations.

In Alpha, objects are defined by the programmer-in-the-small, as well as the programmer-in-the-large. Any notion of inheritance is assumed in Alpha to be handled at compile-time; there are no specific features of the kernel's interface that are meant to support any form of inheritance.

The object model used in Alpha emphasizes a simple and uniform system interface, minimizing the specialized artifacts that are introduced into the (logical as well as physical) programming model. Alpha supports a quite uniform system programming model—everything appears as objects to the programmer. The object abstraction in the Alpha kernel extends to all system services, and encapsulates all of the system's physical resources, providing clients with object interfaces to all system-managed resources (i.e., memory, devices, etc.). This uniform system interface concept allows operations to be invoked on a wide range of entities, ranging from user or system objects and threads, kernel routines, and system hardware.

Objects in Alpha may exist only at a single node at a time; however, objects may be dynamically migrated between nodes. From the kernel's programming perspective, objects exist in a flat universe—i.e., objects are undistinguished by the operating system. Any structure, organization, or discrimination among objects (such as "system" and "application" objects) is imposed by the programmer, and enforced by the kernel.

The objects in Alpha are expected to be medium to large in size (i.e., on the order of 100-10,000 lines of code). This assumption is derived from some practical considerations having to do with the distributed nature of Alpha and the overhead associated with both inter- and intra-node communications (i.e., in order for the system to be practical, the overhead associated with performing operations must be a small percentage of the cost of actually performing the function associated with the operation).

5.1.2 Threads

Threads represent loci of execution control that move through objects via operation invocations. The Alpha thread abstraction is in many ways comparable to the *process* abstraction found in many conventional systems. Unlike conventional processes however, threads move among objects via invocations without regard for the physical node boundaries of the system.

Figure 8 is a snapshot of an example application running on Alpha, consisting of three separate threads in the process of moving through four different objects in the course of performing their computations. Note that the boundaries presented by physical nodes do not appear in this logical view, and both **Thread_a** and **Thread_b** are simultaneously active in **Object₃**.

In Alpha, objects are passive entities while threads are the run-time manifestations of concurrent computations in the system—they are the units of activity, concurrency, and schedulability in Alpha. Threads execute asynchronously with respect to each other, allowing a high degree of concurrency to be achieved, but necessitating that the kernel provide a set of concurrency control mechanisms. These mechanisms allow the necessary degree of concurrency control to be applied at a reasonable cost in terms of overall

system performance (i.e., the mechanisms perform their functions quickly and their use does not seriously restrict concurrency in the applications).

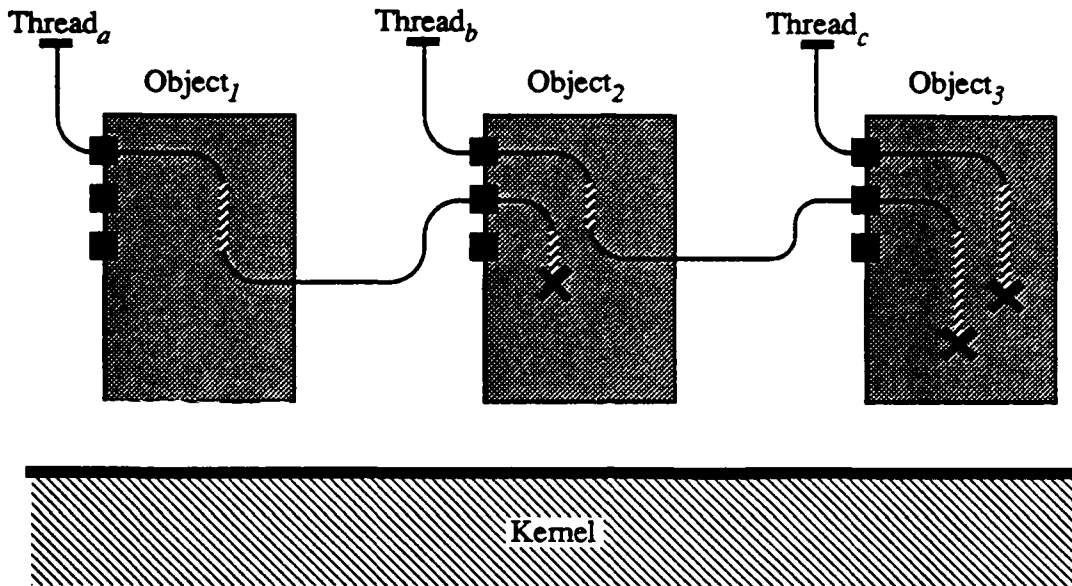


Figure 8: Example Thread/Object Snapshot

The system does not impose an *a priori* limitation on the number of threads that can be executing within an object at any given time. Should concurrency control be required, it is either up to the system level programming interface or the applications programmer to apply concurrency control restrictions (either in a brute-force, or application-specific fashion). However, the kernel provides the necessary (basic) concurrency control mechanisms, and once specified, the system will enforce the desired restrictions on the concurrent execution of threads. In addition, this approach ensures that the thread concept extends naturally to the multiprocessor case (i.e., with concurrency within nodes, as well as among them).

With threads it is possible to implement a wide range of system-level control policies, ranging from low-concurrency structures (such as *monitors*) to medium and high concurrency ones. The thread abstraction simplifies the task of time management in the kernel by being a run-time manifestation of client-defined computations. Threads are also more efficient than most process- and message-based client/server model implementations, because each step in the computation does not necessarily involve an interaction with the system scheduler (for further details, see [Northcutt 88a]).

A significant feature of the Alpha programming abstractions has to do with the combination of the fact that threads maintain a strong correspondence between the program's view of a logical computation, and the system's manifestation of these computations. This feature makes it possible for the client programmer to associate application-specific attributes with computations.

Threads include the local state information for the computations that threads represent. This includes attributes related to the nature of the computation (e.g., reliability requirements, timeliness constraints, and relative importance) that can be used by the system to more effectively manage the system's resources. Threads carry their attributes along

with them as they move through objects (and, transparently, between nodes) in the system. A thread's attributes are modified as it executes through objects, typically in a nested fashion as represented by the thread in Figure 9, which acquires new attributes in the course of its execution (shown both in snapshot and "straightened-out" form).

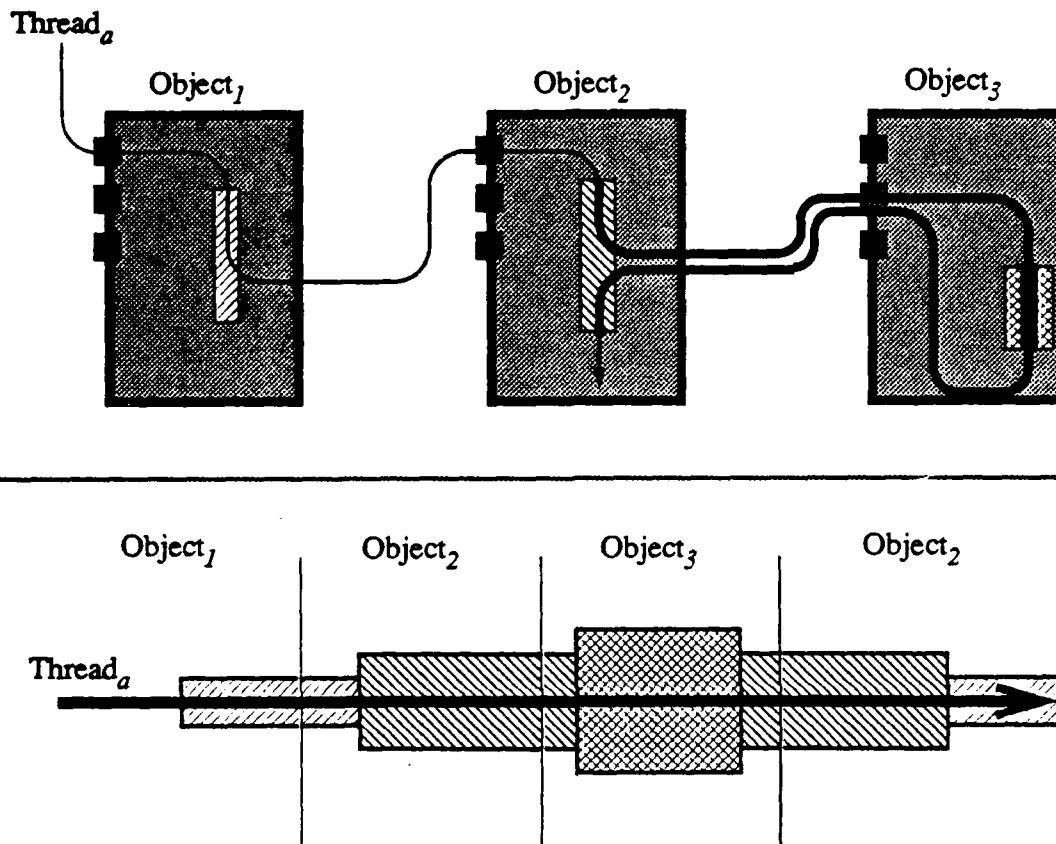


Figure 9: Example of Thread Attribute Nesting

Threads are independent of any specific object, they move among objects, providing animation for these otherwise passive entities, but bear no special association with any object in the system. Furthermore, threads can be dynamically created and deleted in the course of an application's execution. Each time a new concurrent activity is to be performed, a new thread can be created. With Alpha, threads are the form taken by all activity in the system.

From a conceptual standpoint, the Alpha notions of thread and object cleave the point of execution control from code and data definitions in the standard process notion. It is noteworthy that this cleaving of activity from static code/data is true at the implementation level as well—in Alpha, a thread together with an object is implemented in much the same fashion as a typical process.

5.1.3 Operation Invocation

The variety of objects found in Alpha interact most naturally through a form of communication characterized by send/wait interprocess communication. Objects in Alpha therefore use a type of Remote Procedure Call (RPC) for the invocation of operations on objects (as illustrated in Figure 10).

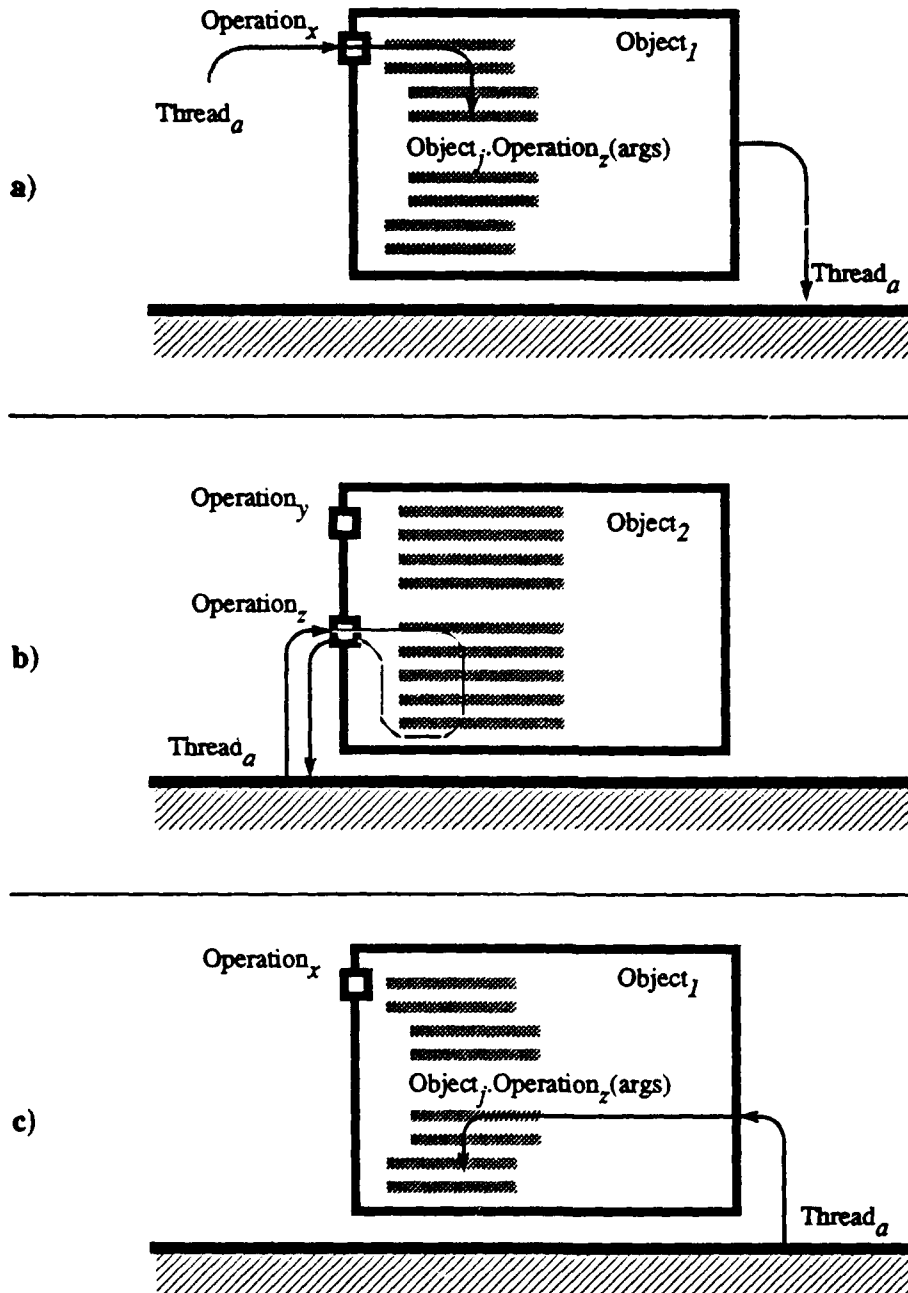


Figure 10: Example Operation Invocation

All interactions with both user and system objects is via the invocation of operations on objects. The operation invocation mechanism is the fundamental facility on which the remainder of the Alpha kernel is based (this is analogous to the role that the interprocess communication facility plays in Accent [Rashid 81]). The operation invocation facility provides simple, uniform access to all objects, whether local or remote, and provides reliable RPC-like semantics. Each invocation returns a success/failure indication (with error code for failures) on its completion. The operation invocation facility does much to mask

the undesirable effects of the system's physical distribution (i.e., node failure, message errors, non-local objects, object migration, etc.), and provides time-driven orphan detection and elimination.

The invocation of operations on objects is controlled by the kernel through the use of a system-protected identifier (i.e., a capability). In this way, the ability of objects to invoke operations on other objects can be restricted to only that set of destination objects explicitly permitted. Capabilities can be given to objects when they are created, or they can be passed as parameters of operation invocations. In the kernel, the capability mechanism provides basic, defensive protection at a low cost in terms of performance.

The operation invocation facility was designed to retain as much of the familiar subroutine-call semantics as possible. The operation invocation facility allows the passing of arguments among objects, with return/value semantics. It is possible to pass simple, or structured data, as well as capabilities as arguments in operation invocations.

Because of the fact that all of Alpha's system services are provided in the form of objects, the operation invocation facility subsumes the role of traditional "system calls," and it constitutes the single trap entry point into the system.

This form of thread/object interaction has a number of good features, including the fact that the RPC style of communication used in Alpha tends to be both simpler and more commonly understood by programmers than more general forms of communication (e.g., asynchronous message-passing) [Nelson 81]. Also, the use of a synchronous form of communication in Alpha does not pose the types of limitations that are typically associated with synchronous communications in process/message-based systems [Liskov 85], due to the nature of the object and thread abstractions. Furthermore, there was initially reason to believe that this type of system could be implemented in a way that is suitably efficient to make possible the construction of meaningful applications programs.

5.2 Time-Value Functions

Alpha uses a novel and highly effective technique for explicitly and expressively manifesting an application's time constraints—as the time-dependent value to the system of completing the computation's activity. The time-driven management of system resources in Alpha depends on the correspondence between the programmer's and system's view of application computations (provided by threads), and application-specified importance and time constraint information (provided by attributes of threads).

The attribute information that is associated with threads in the current implementation of Alpha include: an indication of the relative importance of the thread, the expected completion time for the execution of a region of code; the value (with respect to time) to the system of completing the execution of a section of code; and a probability distribution function that indicates the probability (with respect to time) of completing a region of code. These attributes are associated with threads by parameters specified when creating new threads, and may be modified (at run-time) by invoking operations on the threads themselves.

Collectively, these attributes are the Alpha system's manifestation of *time-value functions*, which represent the value to the system of the execution of a computation (or parts thereof) with respect to time. Time-value functions distinguish between a computation's

timeliness and its importance, it is an aperiodic-based, dynamic approach that is both very powerful and expressive. Time-value functions consist of several different components, as illustrated in Figure 11. The concept of time-value functions was invented by Jensen in 1975 [Jensen 75] and was explored by one of his students working on the Archons project [Locke 86].

I	— thread importance
t_c	— critical time
t_e	— expected execution time
$\sigma(t_e)$	— standard deviation of t_e
S_{pre}	— shape of value function, pre-critical time
S_{post}	— shape of value function, post-critical time
i_{pre}	— importance modifier, pre-critical time
i_{post}	— importance modifier, post-critical time

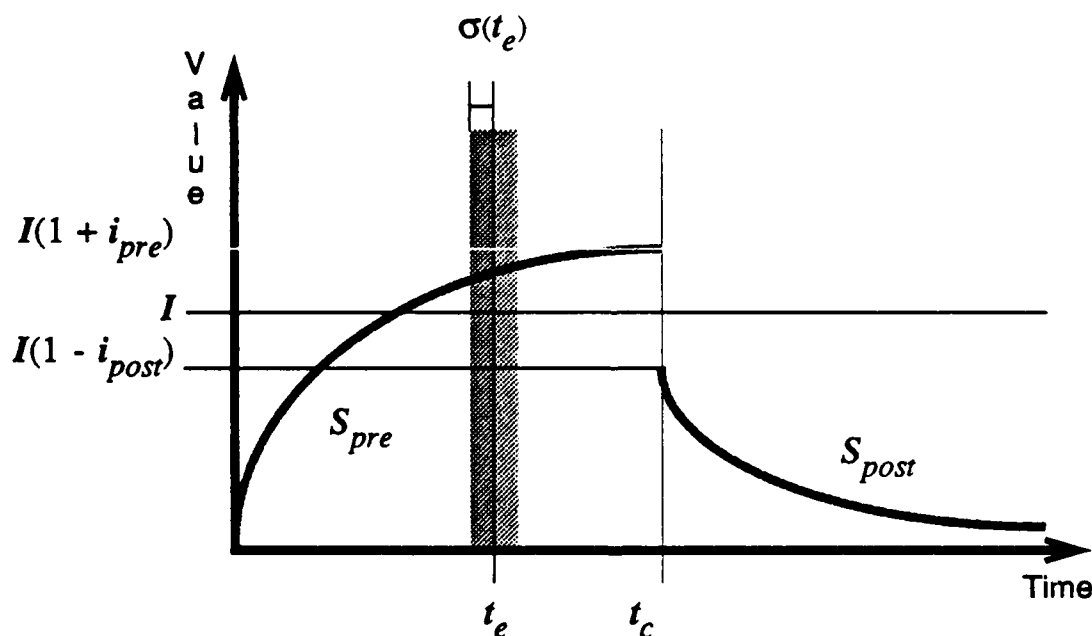


Figure 11: Components of a Time-Value Function

Time-value functions are the basis for resolving all contention for resources in Alpha—e.g., processor cycles; communication management; secondary storage access; synchronization resources (e.g., semaphores). The user-specified time-value functions and the run-time statistics maintained by the system for each thread (such as accumulated execution time) serve as the fundamental inputs to the time-driven resource management policy modules in Alpha.

The time-driven resource management policies in Alpha all follow the same general set of guidelines. The time-value functions for all contending activities are evaluated *collectively*, and then the activities are scheduled so as to maximize the total value to the system for the entire time span they cover. The urgency of the computations requesting system resources is considered first—when there sufficient resources to do so, all of the resource requests are serviced in such an order as to ensure that the computations' time constraints are all met. If there are not enough resources available to satisfy the time constraints of all contending activities, a "best effort" is made to handle the overload condition gracefully (as defined by an application-specified policy [Jensen 76a]). For example, an overload policy might indicate that the system should shed load on the basis of activity importance, or that it should retard all response time performance proportionally to activity importance. The currently preferred policy in Alpha takes the former—i.e., contending requests are selectively denied, on the basis of the urgency and importance of the computations responsible for the resource requests. Further details on this type of resource management is given in Subsection 6.1.3 of this report.

While it is true that higher-utilization techniques impose a performance cost of their own, our studies have shown that in most cases this cost is worth the benefits that it provides [Locke 86]. In fact, it has been shown that many time constraints may not be met under conditions of extremely low utilization [Lechowski 84]. The results of these efforts have illustrated that a large component of the cost can be offset through the use of hardware currency—i.e., a simple, special-purpose hardware accelerator (analogous to a floating-point co-processor). The Archons project's analysis, simulation, and experience indicate that the resources consumed in performing time-value resource management produce greater value for the system than do the simpler techniques which consume considerably smaller amounts of resources that could otherwise be used by the applications. In any event, this project is interested in exploring this research direction in an attempt to develop operating system techniques that will yield both increased system resource utilization and improvements greater than the normal hardware-technology-driven performance increases.

5.3 Implementation Features

The following subsections describe the structure of the software and hardware in the initial implementation of Alpha.

5.3.1 System Software Structure

The Alpha operating system lies in between the application code and the hardware and (in its current implementation) has an internal (logical) structure as shown in Figure 12. The bulk of the functionality described in the preceding sections is provided by the kernel layer, and it is the kernel layer that has received the greatest amount of attention by the project to date. Future work will be directed towards enhancement of the executive and system layers. The following provides a brief description of each of the (logical) software layers in Alpha, including the *Application Layer*, *System Layer*, *Executive Layer*, *Kernel Layer*, and *Monitor Layer*.

The application layer consists of a collection of objects and threads, defined by one or more applications programmers. These objects and threads execute without any special

system privilege (with respect to priorities, access to internals, etc.). The separation of application-level object address spaces is enforced by the underlying hardware.

The system layer consists of a collection of objects and threads, much as found in the application layer (i.e., unprivileged with hardware-enforced separation of address spaces). System-level objects and threads are distinguished from their application-level counterparts only by their access privileges, as defined by the capabilities that they possess. Objects in the system layer provide the higher-level operating system services—e.g., name, authentication, directory, and reconfiguration servers, as well as user interface and programming environment support facilities.

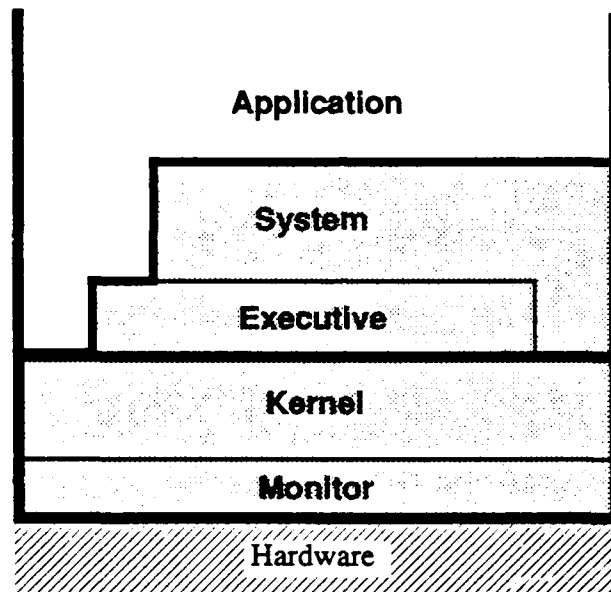


Figure 12: Logical System Software Structure

The executive layer extends the functionality of the kernel by way of a collection of special entities known as *kernel objects* and *kernel threads*. These appear to the programmer who specifies them as normal objects and threads, but they have special characteristics and are implemented in a different manner. Kernel threads and kernel objects are optimizations that permit the system-builder to easily migrate a limited number of threads and objects into the lower layers of the system, primarily for reasons of performance. Kernel object types are specified the same as with normal objects, however they are linked into the system at system-build time and kernel object instances coexist within the kernel's address space. Kernel threads and objects are used to implement system daemons (e.g., interrupt handlers, virtual memory pager, transaction manager, and garbage collector). In addition, the executive layer is where the policy modules that govern the behavior of the local kernel mechanisms reside.

The kernel layer provides the basic system abstractions, upon which the remaining portions of the system are constructed. The kernel consists of:

- system service objects—i.e., routines that provide services to the client by way of object interfaces (e.g., object and thread managers, and semaphore and lock managers)

- hardware supported kernel subsystems—i.e., collections of routines that collectively provide a major internal system facility (e.g., virtual memory, scheduling, communications, and secondary storage)
- the kernel proper—i.e., routines that provide the bulk of the kernel's basic functionality (e.g., operation invocation, trap handling, subsystem interfaces, and physical memory and I/O management)

The monitor layer consists of software that exists within each of the processor's on-board PROM storage. Included in the functions provided by the monitor layer is: low-level I/O support (including the "printf()" routine); TFTP-based boot, upload, and download support; power-on reset initialization and diagnostics; and low-level debugging support.

5.3.2 Testbed Architecture

The Alpha operating system executes on a loosely-coupled collection of dedicated-function multiprocessor nodes, constructed from largely off-the-shelf system components (i.e., the Alpha Distributed Computer Testbed [Clark 83, Northcutt 88b]). The testbed supports the development of software by a collection of system programmers, working from individual (remote) workstations. Furthermore, the nodes of the testbed were designed to allow the exploration of various operating system concepts that may benefit from hardware support.

At a high level of abstraction, the testbed consists of three components—the *development and control system*, the *distributed computer system*, and the *application system* (see Figure 13). The development and control system consists of a collection of Sun Microsystems workstations, running the UNIX operating system, and are connected to the distributed computer system via both Ethernet and 9600 baud serial lines. This part of the testbed is used for the development of application programs, the control of applications running on the distributed computer system, and the monitoring of the operating system as well as application experiments. The software tools that exist on the Sun Microsystems workstations (editor, compiler, linker, window manager, etc.) provide an effective development environment, permitting the testing and debugging of low-level system code by multiple, remote users.

The distributed computer system component of the testbed is the host on which the Alpha operating system executes. It consists of a collection of processing nodes interconnected by a global communications subnetwork. This system is (logically) a single computer, where the network is analogous to the backplane and the nodes being the cards of a conventional processor. The distributed computer is partitionable into separate computers, and interfaces with the outside world via a gateway machine (i.e., a standard Sun Workstation running UNIX).

The application system is comprised of a set of application devices, and is the interface between the distributed computer on which the Alpha kernel executes, and the physical system it is controlling. Application devices can be sources of data to the computer (e.g., sensors) or data sinks (e.g., actuators), and can be a combination of actual or simulated devices. The application system interfaces to the distributed computer system directly through nodes, the network, or gateway machines. Details concerning the Alpha Distributed Computer Testbed can be found in [Northcutt 88b].

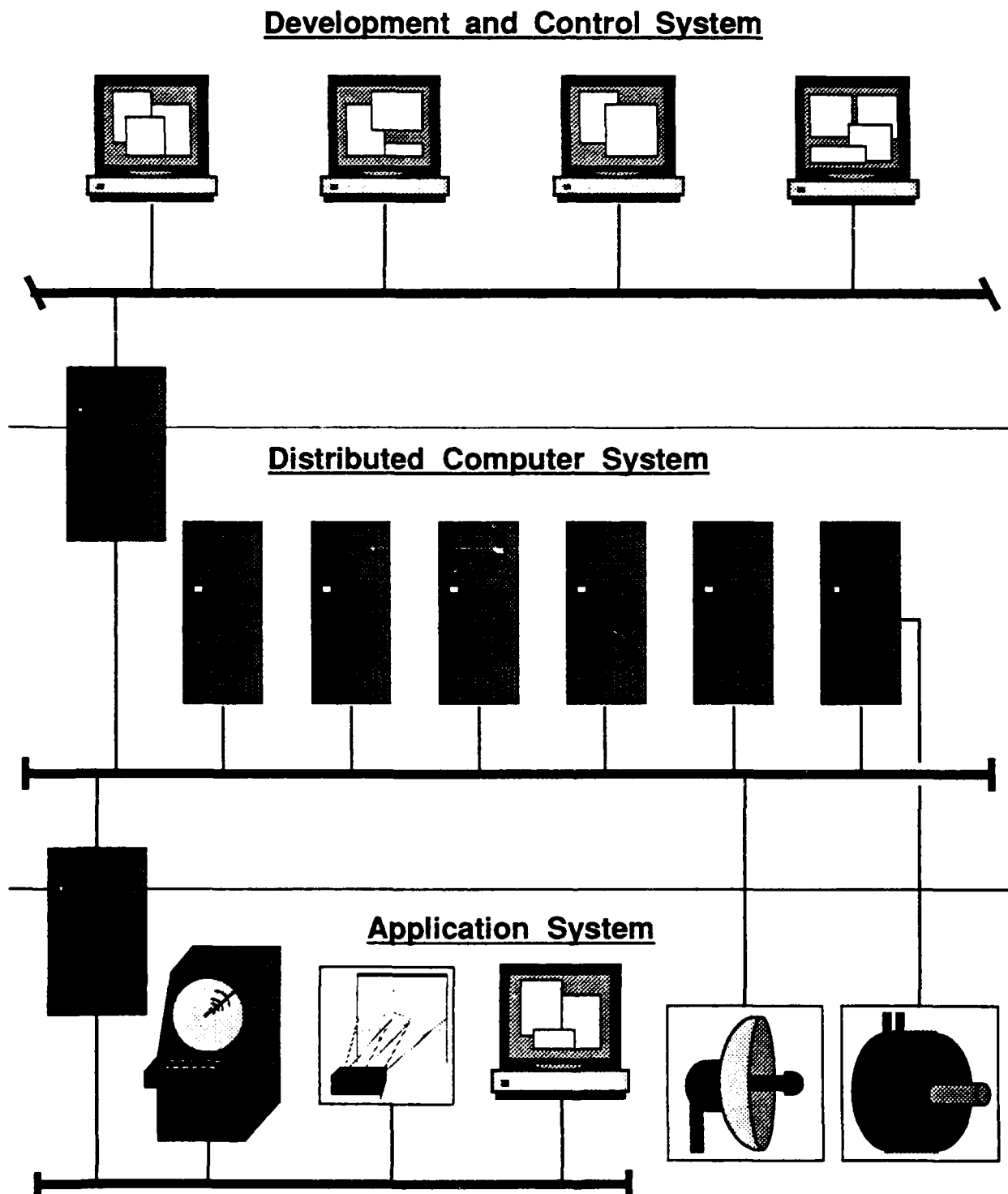


Figure 13: Testbed System Structure

6 Rationale

The following provides an overview of how each of the major issues associated with distributed real-time control systems is addressed by Alpha's kernel mechanisms developed as part of this work.

6.1 Timeliness

Because of the importance of timeliness issues, much attention has been given to considerations of time in the specification, design, and implementation of the mechanisms that make up the Alpha kernel. Most of the common techniques for meeting the (simpler) requirements of real-time systems have been applied in the design and implementation of the Alpha operating system. In addition, each of the facilities in Alpha was specifically designed to help meet the requirements of real-time applications. The general approach to timeliness taken in Alpha is highly adaptive, permits high utilization of system resources, and addresses the problems of timeliness head-on, without making unrealistic or limiting assumptions about periodicity or determinism.

The most visible manifestations of the concern for timeliness in the design and implementation of Alpha are: time-driven resource management, and the use of application-specified time constraints and resource management policies. As defined earlier, the aspect of an operating system design that is most responsible for meeting the needs of real-time applications is the manner in which contention for system resources is resolved; real-time operating systems should take into account timeliness constraints when resolving contention for resources. A guiding principle in the design of Alpha was that all resource management decisions should be made based on the time constraints of the entity making the request for system resources, and when all of the requests cannot be met in a timely fashion, an application-specific overload handling policy is followed in dealing with the requests. To this end, the thread abstraction was developed to provide the framework within which global, time-driven resource management can be performed.

An implication of the use of time-driven resource management in Alpha is that the client must provide the system with a policy for managing its resources (both in normal and overload cases—i.e., when there are insufficient resources to meet the application's needs). In addition, the client must provide the system with the information needed to manage each of the application's time-critical computations (e.g., the timeliness constraints and relative importance of each computation). The Alpha operating system was designed to allow the system's clients to provide custom resource management policies (e.g., for application processor scheduling and virtual memory paging), and to allow the simple and natural expression of application-specific time constraints (by associating timeliness attributes with threads).

The influence that the requirements of real-time programming had on the design and implementation of Alpha is evident in the system's basic abstractions, programming interface, kernel subsystems, and kernel mechanisms.

6.1.1 Effects on the Basic Abstractions

The Alpha fundamental programming abstractions were specifically designed to support the overall objective of global, dynamic, time-driven resolution of contention for system

resources (e.g., processor cycles, communication bandwidth, memory space, or secondary storage). In particular, the thread abstraction provides a framework for injecting the application's time constraints into the system, and a basis upon which application-specific system resource management policies can be defined.

6.1.1.1 The Thread/Object Approach

Threads maintain a direct association between the programmer's logical view of each concurrent stream of execution that makes up an application, and the system's physical manifestation of these logical computations. Thus, threads provide a direct means for associating the timeliness requirements that clients specify for their computations, with the specific, location-transparent, run-time entities that the kernel manages. In this manner, global importance and urgency characteristics of computations can be propagated throughout the system and used in resolving contention for system resources according to client-defined policies.

In Alpha, all requests for system resources can be associated with the computation that made each request, and the system uses these attributes to service the requests according to the application-specified management policy for that resource. For example, the current scheduling policy in Alpha sequences the execution of threads in such a fashion as to allow all of the threads' specified time constraints to be met (whenever possible), using the (time-varying) urgency and importance attributes of threads that are ready to run.

The thread abstraction in Alpha was not designed to meet the requirements of timeliness in isolation, but the requirements of distribution, robustness, and adaptability were also considered in the design. Threads provide a unified means of managing all resources in the system—both within and among the processing nodes in the distributed system. The application-specified timeliness attributes of computations are carried with threads as they move through objects and across the system's nodes. This allows a globally consistent form of distributed resource management to be provided through a form of implicit coordination (i.e., at each node, the same resource management policies are applied to the threads' global attribute information).

The Alpha thread/object abstractions are better suited for meeting the needs of distributed real-time control systems than the conventional process/message abstraction. The main reason for this is that the Alpha thread/object abstraction maintains a strong correspondence between the logical and physical views of computations, while typical process/message-based systems do not maintain such a strong relationship. This close association in Alpha allows requests for system resources to be tied to the logical computations (that the programmer associated attributes, such as time constraints, with) to allow the time-driven management of resources.

6.1.1.2 Comparisons to Traditional Approaches

To illustrate some of the significant differences between Alpha and traditional process/message-based systems, consider a simplified application consisting of three separate computations, each of which has a portion in common with the others (i.e., they make use of some common service, or they perform the same sequence of instructions). In most systems, the functionality of an application is decomposed into software modules

for reasons of intellectual manageability, reusability, relocatability, and concurrency. In the process/message approach, the software modules that are combined to make up computations are known as *processes*, while in Alpha, the programming modules that define computations are known as *objects*. In a normal process/message-based system, these logical computations are mapped onto collections of processes that interact by message exchanges, while in Alpha each computation is defined by a thread that moves among a set of objects via operation invocation.

Figure 14 illustrates a stylized process/message-based implementation of this example application, where processes 1, 2, and 3 obtain the service provided by process *x* through the exchange of messages. Figure 15 provides an illustration of the same application implemented in Alpha, where threads *a*, *b*, and *c* begin execution in objects *i*, *j*, and *k* (respectively), and obtain the service provided by object *x* by invoking operations on it.

A number of significant differences exist between these two approaches. For example, in the process/message case there is a discontinuity between the programmer's logical concept of three application computations, and the four interacting processes that the system supports in its implementation. This discontinuity requires that the programmer perform a transformational step in going from the application's conceptual design to its realization (while the transformation is not significant in this example, it can be quite significant for less trivial applications). Furthermore, the discontinuity between the application programmer's and system's views of computations interferes with the system's ability to perform global time-driven resource management. To illustrate this point, note that in Figure 14, each computation is being performed by the cooperative interaction among a pair of (as far as the system is concerned) independent processes, and (as is typical for such systems) each process is scheduled based on its own, static, priority. In this case, the system manages resources based only on the requests it receives from individual processes, and not based on the characteristics or requirements of the logical computations. The result of this is that process *x* executes at its given priority level, regardless of on who's behalf it is performing its function, so the portion of the first computation being provided by process *x* is executed at too low a level of priority, and similarly part of third computation runs at a higher level of priority than it should. This example, and various schemes that have been developed to help processes deal with these problems (e.g., priority propagation) are dealt with in Chapter 6.

In Alpha, each thread represents an individual computation and moves among objects independently, carrying with it the attributes assigned to the computation by the application programmer. This approach maps the logical computation directly onto its system manifestation and allows the system to receive resource requests from the computations themselves and not some unrelated artifact. The result of this is that in each of the threads in Figure 15 execute in their proper order (based on their application-specified urgency and importance attributes), regardless of the object they happen to be executing in at a given time. In effect, the server's function is performed, not only with the attributes of the individual computation requesting the service, but actually by the computation itself.

An additional benefit of the Alpha thread/object abstractions is that they admit of an implementation that does not involve unnecessary scheduler interactions on each instance of inter-module communication. In typical process/message-based systems, communication and scheduling activities are intertwined in an undesirable fashion—

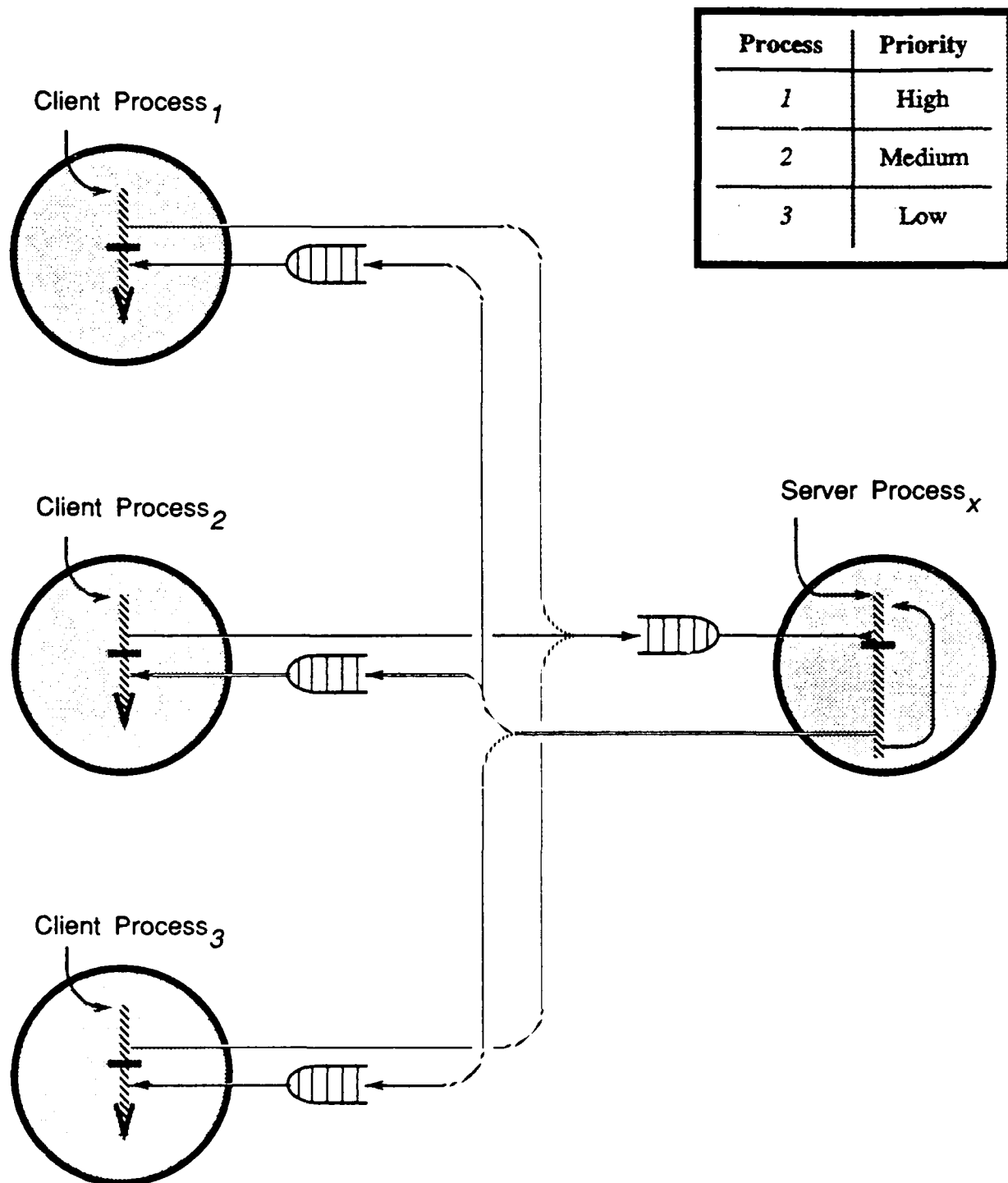


Figure 14: Typical Process/Message Interactions

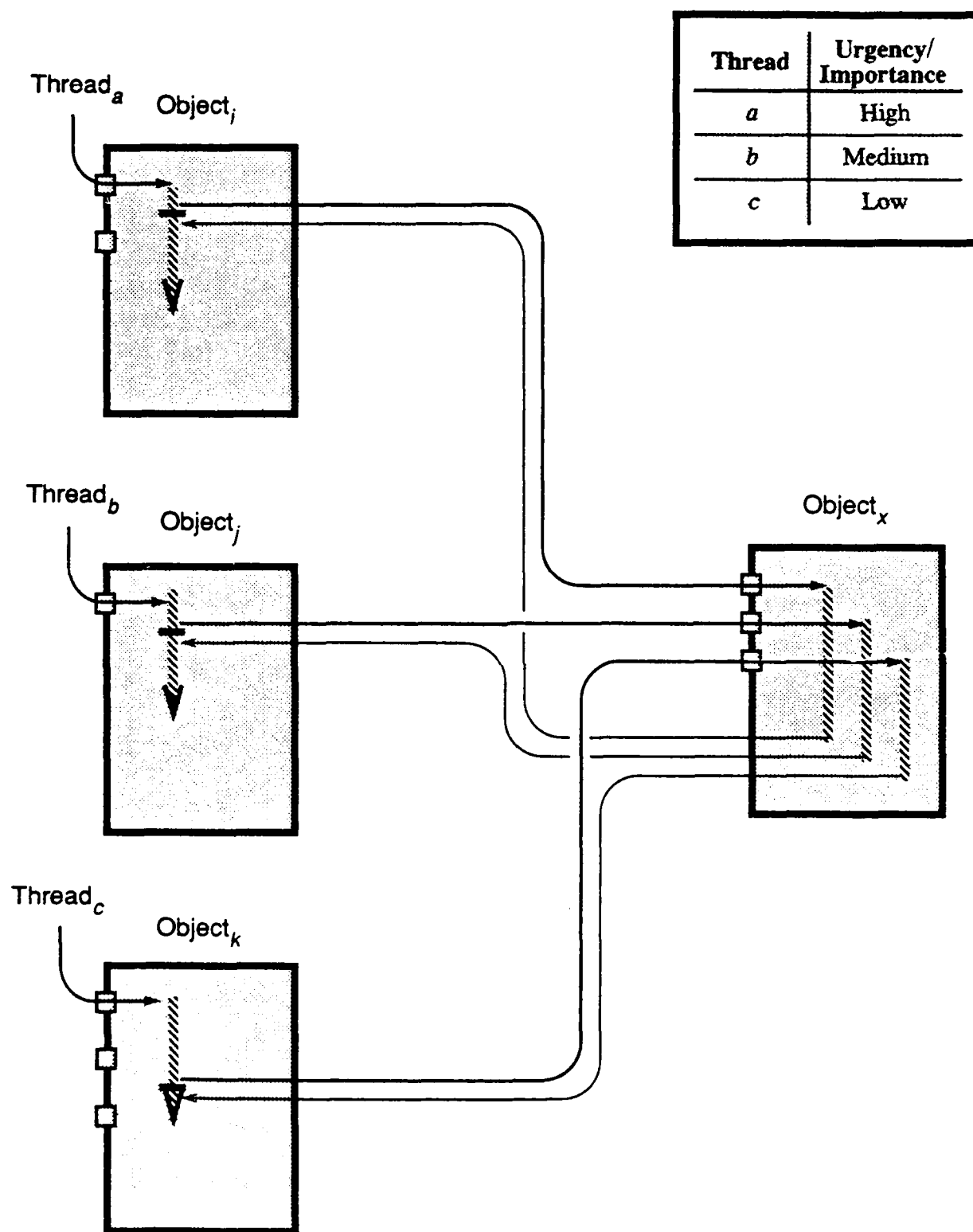


Figure 15: Typical Thread/Object Interactions

to accomplish the next (logical) phase of a computation requires a scheduling activity. This is an artifact of the process/message approach, because communications itself is not a scheduling event. When the system's scheduler has determined that a computation should receive processor cycles, the computation should retain the processor until a legitimate scheduling event occurs, and not relinquish the processor whenever it wishes to execute another part of the computation. This problem is intrinsic to these styles of programming—in the Alpha version of the example, there is a single schedulable entity for each computation (i.e., the threads), while in the process/message example there are multiple schedulable entities involved (i.e., the processes).

While the emphasis in this example was on the management of processor cycles (i.e., scheduling), the Alpha programming abstractions allow for similar time-driven management of any I/O, communications, or memory resources required by the computations. In most process/message-based systems the management of these resources interact in an arbitrary (and largely uncontrollable) fashion. Alpha threads provide a unifying means of managing system resources in a consistent fashion both within and among the system's nodes.

6.1.2 Effects on the Programming Interface

The Alpha system's timeliness requirements have had a very dramatic effect on the system's programming interface (i.e., the interface that the kernel provides to its clients). In particular, the use of time-driven resource management demands that the programmer be able to provide the system with the information needed to carry out the desired resource management policy. Furthermore, the very fact that time constraint attributes may be specified for computations implies a need for the programmer to be able to deal with the effects of missing hard deadlines (i.e., time constraints that define a time after which the computation no longer has positive value to the system).

6.1.2.1 Specifying Timeliness Attributes

The Alpha programming interface supports the notion of *time constraint blocks*, that provide a convenient way for the programmer to define time constraints for the execution of specific regions of code within objects. Time constraint blocks allow the programmer to associate dynamic time constraints with blocks of code by, in effect, placing brackets around a region of code. Alpha supports the application of multiple, nested time constraints to threads, where all of the time constraint blocks in which a thread is active apply at all times (i.e., a nested constraint does not invalidate those enclosing it).

Alpha provides the programmer with flexibility in determining the binding time for the parameters of time constraint blocks. For example, the time constraint block parameters may be defined at compile time (i.e., *early binding*), or they may be defined whenever a thread enters the defined block of code (i.e., *late binding*). Early binding is useful where it is desired that the code enclosed by a time constraint block must be executed by a certain (constant) time, regardless of which thread is executing the instructions (e.g., when the combination of reading a sensor and moving an actuator must be performed within a certain amount of time). Late binding is useful where a block of code has a time constraint, but exactly what the constraint is must be computed each time before entering the block (e.g., when the time required to complete a testing operation is dependent on the velocity of a sample past a sensor). Furthermore, there are different types of late bin-

ding that can be performed. The parameters for a time constraint block may be a function of the global state of the object alone, in which case the timeliness constraint would be the same for any thread executing the time constraint block. Alternatively, the time constraint block parameters may be a function of the local state of a thread (i.e., stack variables or incoming parameters), in which case the timeliness constraint for the block of code could vary with the thread executing within it. This flexibility allows the programmer to more dynamically and accurately represent the timeliness needs of his computation to the system, thereby making it possible for the system to manage its resources more effectively.

6.1.2.2 Handling Expired Time Constraints

When a thread's time-value function is no longer positive (i.e., the thread has failed to complete the execution of a time constraint block within the specified amount of time), the thread's execution within the time constraint block should be terminated. This is because the thread's time constraint cannot be met, its continued execution and consumption of resources interferes with the successful meeting of other threads' time constraints, and the work that the thread continues to do represents more effort that must be expended to compensate/undo to make the object consistent.

The fact that there may be considerable negative value to the application of the continued execution of threads whose time constraints cannot be met demands that the system must put a stop to the execution of such threads in a prompt fashion. Simply halting a thread when its time constraint has been missed is a totally unacceptable solution—in a real-time system, missing time constraints is as much a part of life as defining and meeting time constraints. A much more meaningful action to take when a time constraint expires (e.g., a deadline is missed) is to redirect the execution of a thread to an exception handler specified by the programmer, where the programmer can define what should be done. The programmer might want to perform one or more of these actions: kill the thread, put the object back into a consistent state, report an exception to another object, compensate for the partial execution of the block, retry the block, log an event, or just proceed on.

Alpha supports the clean-up of computations which fail to satisfy their time constraints, to avoid wasting resources and executing improperly timed actions. This is done by immediately and forcibly diverting the normal flow of the thread's control to a defined exception handling block of code (as illustrated in Figure 16). By providing the programmer with a way of going to a defined point in his code on exception with all of the state information of the computation, a wide range of user-definable exception handling policies can be implemented. This is to say, a time constraint block has a single entry point, but has two exit points—one for normal exits (i.e., the necessary computation was completed before the specified time), and one for abnormal exits (i.e., an abort induced when it becomes known to the system that the computation cannot be completed in the required time). A thread exits a time constraint block for one of three reasons: the thread has completed executing the code (and is exiting normally); the thread was aborted pursuant to a resource management decision by the scheduler (e.g., a missed time constraint); the thread was aborted for some other reason (e.g., it was in an atomic transaction that has been aborted).

A time constraint is valid as long as the thread is executing code within a time constraint block. This implies that once a thread's execution is directed to a time constraint block's abort code, the block's time constraint is no longer in effect and the thread assumes attributes of its next outermost time constraint block. In this way, all execution of exception handling code is done under the proper computational attributes (i.e., the system pops to the next level of time constraints so that the thread's attributes properly correspond to the thread's new state). This means that the system continues to manage resources in the desired fashion, regardless of expired time constraints. The manner in which exceptions are handled by time constraint blocks is largely the same way that block-structured, nested, mechanisms are used for handling other types of exceptions in the system (e.g., atomic transactions).

The kernel can also indicate, at run-time, the probabilities of successfully meeting a given time constraint. This feature provides an early indication of an exception and permits the application to determine the proper course of action for each individual case.

```

/* An Arbitrary Object Type Specification */
OBJECT Obj_A() {
    /* the object's first operation */
    OPERATION Opr_1(parms) {
        /* start of the operation's code */
        ...
        /* enter this operation's first time constraint block */
        TIME_CONSTRAINT_BLOCK(parm1, parm2, parm3) {
            /* start of code that is to be executed under this time constraint */
            ...
            /* an arbitrary conditional statement */
            if (condition) {
                ...
                /* enter a nested time constraint block */
                TIME_CONSTRAINT_BLOCK(parm1, parm2, parm3) {
                    /* code to be executed under the nested time constraint */
                    ...
                }
                ON_ABORT {
                    /* code to be executed if the time constraint is missed */
                    ...
                } /* end of the nested time constraint block */
            }
            ...
        }
        else {
            ...
        }
        ...
    } /* end of the first time constraint block */
    ...
    /* enter this operation's next time constraint block */
    TIME_CONSTRAINT_BLOCK(parm1, parm2, parm3) {
        /* start of code that is to be executed under this time constraint */
        ...
    } /* end of the first time constraint block */
    ...
} /* end of the first operation */
} /* end of the object */

```

Figure 16: Example Use of Time Constraint Blocks

6.1.3 Effects on the Kernel Subsystems

Each of the major kernel subsystems (i.e., the scheduling subsystem, the communications subsystem, and the storage management subsystem) was affected by the system's timeliness requirements. The following paragraphs give an overview of the major effects that timeliness had on the design and implementation each of the subsystems.

6.1.3.1 Scheduling Subsystem

The primary system resource that any operating system manages is application processor cycles, and in Alpha this resource is managed by what is known as the scheduling subsystem. The subsystem was designed and implemented to support: a wide range of different scheduling policies, the management of multiple applications processors, and the computation of schedules concurrently with respect to the execution of application code. In the current implementation, the scheduling subsystem executes on a separate processing element within a node, and interacts with the kernel proper (executing on an applications processor) via the kernel's inter-processor message-passing mechanism. In Alpha, per-node scheduling is done at the kernel level, while global scheduling is performed at the system level.

The scheduling subsystem on each node maintains information on the time constraints of all threads currently active on that node, as well as their current state with respect to those constraints (e.g., the thread's accumulated execution time, whether the thread is blocked, and what system resources the thread has acquired the use of). The scheduling subsystem on a node generates a new thread execution ordering (i.e., schedule) on each scheduling event, and causes an applications processor to perform a context swap and dispatch a new thread when the currently executing thread is no longer at the top of the schedule. Scheduling events occur when a thread: is created/destroyed, unblocks, blocks, is run, is preempted, changes importance, enters/exits a time constraint block, handles an exception.

While many different types of scheduling algorithms may be (and have been) inserted into the framework provided by the scheduling subsystem, the algorithm that has received the greatest amount of use and attention to date is known as the *best effort* scheduling policy. The best effort scheduling algorithm is an outgrowth of the Archons project's long-standing research efforts in real-time scheduling and its use in Alpha represents the first practical application of our research work in this area. The best-effort policy takes application-specified information concerning the timeliness constraints of applications (i.e., the threads' attribute information), and attempts to meet all the application's time constraints, adapting to unexpected events. When the demands for processor cycles exceed the available supply (i.e., not all time constraints can be met), the best-effort policy discards requests (i.e., omits ready threads from the scheduling list) in such a fashion as to maximize the expected value to the system of the threads that are being scheduled for execution.

The best-effort policy includes a particular overload handling sub-policy, however examples of other overload sub-policies include: discarding the least important threads from the set of ready threads; discarding threads to maximize the total number of threads whose time constraints are met; not discarding any threads, but instead have all threads be tardy by some average amount of time (i.e., distributed the overload evenly across all ready threads); or do not discard threads, but have the ready threads be tardy by an amount inversely proportional to their value to the system.

It should be noted here that the best-effort scheduler implemented in Alpha can be made to behave like a range of different schedulers, depending on the amount and type of information given to it by the application programs. For example, with the thread's critical time parameter alone, the best-effort scheduler can behave like a *deadline* or a *rate*

monotonic scheduler. When given the expected execution time parameter alone, the best-effort scheduler behaves like a *shortest-processing-time-first* scheduler. With nothing but the thread's importance parameter, the best effort scheduler degenerates to a *priority* scheduler. Finally, without any of the thread's scheduling attributes at all, the best-effort scheduler runs threads like a *round-robin* scheduler.

With the full information provided by the applications programs with time-value functions, the best-effort scheduler can manage applications processor cycles much more effectively than can other scheduling algorithms. Time value functions effectively distinguish between the urgency and importance of a computation, whereas these attributes are most commonly encoded into a simple, small-integer priority code in existing systems. Also, the combination of time-value functions and threads allows a direct expression of an application program's time constraints, with no translation or transformation required between the specification of an application computation's timeliness requirements and the computation's physical implementation. Time-value functions are a versatile representation that can uniformly (i.e., without special cases or exceptions to the model) express many different types of timeliness constraints, including: hard deadlines, soft time constraints, hard or soft execution time windows, and delayed execution. Because time-value functions can be specified dynamically (and the time constraint parameters may be run-time variables), a time constraint applied to a particular section of code can, over time, exhibit a varying degree of "hardness" (i.e., the value of continued execution following the computation's critical time may change) and a dynamic maximum value (i.e., the global value to the system of completing the computation may change with respect to time).

Other, more typical, scheduling algorithms do not attempt to deal with overload conditions properly (in fact, some policies, such as deadline, exhibit particularly bad performance in overload cases). In recognition of the system's robustness requirements, the best-effort policy was designed to handle overload conditions gracefully, attempting at all times to maximize the global value (as defined by the application) of the execution of computations to the system.

The best-effort algorithm in Alpha was designed from an aperiodic point of view, in that no special case treatment is given to events that occur cyclically. Many real-time schedulers use the cyclic nature of some applications to obtain analytical leverage in making specious claims of "guarantees," and aperiodic events are force-fit into the periodic mold. Alpha makes use of a more general solution where time constraints are considered in a uniform manner, and no special significance is given to an event that may happen to recur in a more or less regular time pattern. Cyclic events are "unrolled," with each iteration treated as a independent instance of a time constraint applied to a section of code. This allows periodic and aperiodic activities to be handled in an integrated, uniform manner and yields a highly adaptive scheduling subsystem. With the best-effort scheduler, the system does not fall apart when an assumption concerning the periodicity of events is violated, nor does it require a reevaluation of all of an application's timeliness constraints when a change is made in one section of code.

Furthermore, in Alpha there is no premature binding of timeliness attributes that would restrict the ability of the system to carry out certain policies or its ability to adapt to events that might occur between the time that the binding is done and when the system

makes resource management decisions. For example, the timeliness attributes are maintained with threads in their "raw" form and are not transformed into some compressed form such as priorities, meaning that there is no loss of information due to transformations of timeliness attributes by the system. Furthermore, because the timeliness attributes of threads can be dynamically modified throughout the course of its computation, the system can be more adaptive than those which statically define time constraints for computations.

The best-effort scheduler allows the Alpha system to be employed in a more conventional manner that guarantees real-time constraints can be met for low-level static applications (e.g., rate-monotonic or rate-group scheduling). However, Alpha's best-effort scheduler has significantly greater capabilities for more general real-time command and control applications than is available in typical real-time schedulers.

6.1.3.2 Communications Subsystem

The demands of timeliness are also reflected in the design and implementation of the Alpha communications subsystem. This subsystem manages the resources associated with the communications between nodes in the distributed computer system, including interconnection bandwidth and the processing cycles and memory associated with the communications activity. These resources can be managed in the same fashion as the scheduler uses to manage application processor cycles—i.e., trace each request to a thread and use its attributes to help meet the desired time constraints and to resolve conflicts when contention for resources occurs. In this way, the communications resources can be managed in the same fashion as all others, using the same information and the same policies.

The current testbed hardware does not include an interconnection network that uses a priority-based bus arbitration scheme. However, if such a bus were used, this notion could carry the desired resource management policy all the way down to the bus-arbitration level. For example, the time-value information is used to provide the most important activity on each node with access to the bus, the bus arbiter could then be used to allow the most important activity across all of the nodes to use bus next[†].

6.1.3.3 Storage Management Subsystem

The storage management subsystem also reflects the timeliness requirements in the manner in which it manages the system's secondary storage and virtual memory resources. The storage management system makes use of the attributes of the threads that are responsible for storage resource requests—e.g., page-in requests, physical memory page allocation requests, and write page to stable storage requests.

The storage management subsystem also maintains the run-time statistics that are required to determine the working set of pages needed by each thread executing within a particular object. The storage management subsystem uses the client-specified attributes of threads in conjunction with its own run-time statistics to manage storage

[†]Note that this is only an issue when it is necessary to obtain extremely high utilization from the bus and the non-preemptible period of bus usage is significantly larger than that which is typical of the computing speed versus communication speed ratios that exist in products today.

resources (i.e., virtual memory, primary memory, and secondary storage) according to a given policy, just as the scheduling subsystem does for application processor cycles. For example, the virtual memory policies can use application-specific information about individual threads to do pre-paging and page victimization in a timely fashion—e.g., do not victimize pages needed by threads with short deadlines, order the servicing of pre-paging requests based on the time constraints of the requesting threads, and order the disk read and write request queues based on the time constraints of the threads associated with the blocks being read and written.

6.1.4 Effects on the Kernel Mechanisms

The design and implementation of the Alpha kernel's mechanisms provide further support of real-time applications. For example, the synchronization primitives are designed to consider timeliness constraints in their function; each time a thread performs a synchronization operation (i.e., manipulates a semaphore or a lock), the scheduler is notified. When the time comes to unblock a thread waiting for a synchronization token (e.g., when a V operation is done on a semaphore, or when a lock is released), the timeliness attributes of each of the blocked threads is considered and the thread with the most demanding time constraint is chosen. Furthermore, the timeliness attributes of all of the threads waiting for a synchronization token held by a thread is considered by the kernel when determining when a thread should be preempted. In some cases, it is desirable to allow a thread with less stringent timing constraints to execute before one with tighter constraints in order to allow free up a thread with even greater urgency that is waiting for a synchronization token to be released (for examples of this, see [Northcutt 88a]).

The Alpha operating system was designed so that the execution of threads can be preempted while they are executing within the kernel. This is as opposed to the case in some operating systems (e.g., UNIX) that disable preemption within the kernel. This is done to reduce the amount and complexity of synchronization required by the system to maintain the consistency of its internal data structures. However, an application typically spends 40%-50% of its time executing within the operating system's kernel, and so a system's responsiveness to the dynamics of real-time applications suffers if preemption is not permitted while a computation's point of control is within the kernel.

Another example of the influence that the system's timeliness requirements had on the design and implementation of Alpha is that (to the greatest extent possible) all system activities are implemented with threads. This was done because of the observation that there are many activities in typical operating systems that fall outside the realm of normal, client computations, and as such are not governed by the system's normal resource management policies. For example, interrupts, DMA-type activities, and (hardware) exception handling are all activities that consume (or "steal") system resources (e.g., processor cycles, memory, and I/O bandwidth), but are usually not managed by the operating system. By converting all such activities into threads (e.g., all interrupts are converted almost instantly into threads—buffer data, reset device, and unblock a thread), the system can manage more of its resources according to their given time constraints, in a uniform and globally consistent fashion, according to the application-defined policy.

In addition, the Alpha kernel mechanisms were designed and implemented so as to ensure that they will not require highly variable or unbounded amounts of time to com-

plete their functions. This is meant to enhance the predictability of behavior, as is frequently desired of real-time systems. Another aspect of this is manifest in the application of optimizations that were directed towards the exception rather than the normal cases (examples of which are provided in the following sections).

6.2 Distribution

One of the basic assumptions for Alpha was that it would execute on a physically distributed computing system, for reasons of both robustness and performance. A distributed system can provide concurrency of execution among (as well as within) its constituent nodes, and the bottlenecks of traditional centralized systems can be avoided. Distributed computer systems mirror the physical distribution of real-time command and control applications by placing processing elements in near proximity to the sources and sinks of application information. This results in a reduction in communications delays between processing and data source/sink sites, and allows the specialization of processing nodes for increased overall system cost-effectiveness. Physical distribution of a computing system can also improve its survivability by providing independent failure modes, greater isolation of faults/damage, and improved support for graceful degradation of function and dynamic reconfiguration.

To obtain the benefits offered by distributed systems, a number of additional functions must be performed and additional features must be dealt with that either did not exist in centralized systems or could be safely ignored. Either the system software or the application programmer must perform inter-node resource management and deal with the effects of physical distribution. Should this be done by the application programmer, the effort is recurring, possibly non-standard, and frequently non-expertly implemented. It is furthermore difficult, or impossible, to coordinate the resource management functions performed by the clients with that (largely local functionality) provided by the underlying system software.

The Alpha operating system deals explicitly with the effects of distribution by means of a facility for efficiently providing reliable, physical-location-transparent communication at a low level in the system—i.e., the operation invocation facility. The operation invocation facility is the primary kernel service upon which all other abstractions depend. By making the invocation of operations on objects reliable and location-transparent, the effects of physical distribution are reduced, in effect, to the semantics of procedure calls that return an indication of the success or failure of the invoked operation. Furthermore, by having all objects (and even mechanisms within the kernel) use the invocation mechanism, it is possible to enforce a uniform access method to all system resources, regardless of their actual location within the system. System resources can therefore be managed and accessed uniformly regardless of their physical location.

The operation invocation facility represents the single focal point of all interactions among objects, as well as between objects and the kernel. This provides a convenient point where system access control and data format translation functions can be performed. Alpha has attempted to eliminate any alternate communications channels (such as shared memory) that might be used by programmers and would inhibit the system's ability to perform dynamic reconfiguration. Also, the full visibility of all instances of inter-object communication makes possible a number of system resource management optimizations that make use of object/thread interaction patterns.

The kernel provides mechanisms to permit the programmer to deal with node and communications network failures (i.e., the operation invocation facility), and to dynamically (and transparently) migrate objects among nodes. In addition to the operation invocation facility, many of the Alpha operating system's other mechanisms are designed to cope with the effects of the system's physical distribution.

The Alpha system abstractions do not directly reflect the (significant) performance differences in interactions between objects co-located on a node, and objects on different nodes. Whereas some operating systems introduce sub/super-structure abstractions that recognize and exploit the differences between local and remote objects, in Alpha the intent was to maintain a uniform programming abstraction that projects to the programmer an image of a (logically) centralized system. Should additional structure among the programming entities be desired, it can be applied at the system or applications level. This approach was taken with the desire that the system software abstract away the undesirable aspects of physical distribution, and the kernel-provided programming abstractions maintain a strong relationship to their implementation. In some systems, there are sub-computational entities that are a part of the programmer's abstractions but are not visible to the underlying system software. This approach suffers from the fact that there is a loss of correspondence between the logical and physical aspects of computations; the system manages a single entity, which encompasses more than one of the programmer's computations. In such cases, the programmer's individual computations cannot be independently relocated across the distributed system at run-time. A further implication of an approach where the programmers' abstractions differ from their system implementation is that the system is unable to distinguish among the requests for resources it receives and so it cannot perform its management functions effectively (e.g., when one sub-entity generates a page-fault, all of the related entities are blocked).

6.3 Robustness

For the purposes of this work, a commonly used distributed system failure model was adopted [Anderson 81]: the types of failures considered here are the failures of both hardware and software components, in both the system and application domains, including both *transient* and *hard and clean* failures.

The robustness techniques employed in Alpha are supported primarily by kernel mechanisms that provide a client interface at which failures in the underlying system are abstracted into a set of well-defined, predictable behaviors. In particular, the following robustness issues are addressed:

- **consistent behavior of actions**—provided by mechanisms that support (independently) the attributes associated with atomic transactions (i.e., atomicity, permanence, and serializability). These attributes are provided in the form of individual mechanisms in order to provide a range of levels of service at a range of costs, allowing applications to pay only for the amount and type of reliability needed.
- **availability of services**—provided by mechanisms that allow objects to be replicated and manage the different types of interactions defined on those replicas.

- **graceful degradation**—provided by mechanisms that use an ordering function (currently based on the timeliness constraints and relative importance) associated with all requests for services, in order to sacrifice lower-valued requests in favor of higher-valued ones when resource allocation conflicts arise.
- **fault containment**—provided by mechanisms that place each object in a separate (hardware-enforced) address space, and by separating software components into private system-enforced protection domains, with all interactions restricted to those explicitly allowed by the capability mechanism. This supports a form of defensive protection, where errors are prevented from propagating among objects.

While the Alpha operating system provides a set of mechanisms to support these objectives, its robustness mechanisms are not intended to form a complete facility. The kernel is intended as a framework within which policy issues relating to these robustness techniques can be explored. The mechanisms provided in the kernel for atomic transactions and replication are initial versions of the more complete sets of mechanisms being developed in these on-going projects. The on-going work by the Archons project is addressing the development of system-level policies which use these mechanisms. In fact, the area of real-time atomic transactions is addressed by an in-progress thesis research project [Clark 88].

The Alpha operating system's concern for reliability is manifest at all levels within the system—from the basic assumptions, to the programming abstractions, and all the way down through the system's design and implementation. The variety of object-orientation in Alpha was chosen in the belief that it would be well suited to the type of robustness techniques that have been developed by the Archons project for real-time command and control applications [Clark 88, Sha 85]. The Alpha object model provides a simplified (or constrained) control structure for interactions among software components (as compared to a more general process and message-based system model) and restricts the accessibility of the encapsulated data items. Among other things, this serves to simplify the task of tracking the operations performed on objects that is required in the implementation of atomic transactions. The fact that the object model centralizes all access to encapsulated data reduces the complexity involved in structuring operations so as to maximize the concurrency that can be obtained from objects (both within and outside of atomic transactions).

6.3.1 Exception Handling

The different types of exceptions that can occur in Alpha are classified into three groups: *client-defined*, *kernel-defined*, and *machine-defined* exceptions. Client-defined exceptions indicate client-level events by way of the system's normal operation invocation facilities. For example, the failure of a client operation may be indicated to the invoking object by way of an operation invocation return parameter. Also, objects can "register" operations with other objects that are to be invoked when a given client-level exception occurs. In this way, a thread can invoke a given operation on an object when a client-level exception is detected, which may change the state of the object in such a way as to allow other threads to detect the exception (i.e., an implementing an interrogative signal, as opposed to an imperative one). These exceptions exist at the application-level

and system-level in Alpha and the kernel is typically not involved in the signalling or interpretation of this class of exceptions. However, the application programmer is capable of generating other types of exceptions with his code—e.g., divide by zero or invoke an abort operation. Even though an explicitly generated operation, like a transaction abort, is not strictly an exception, its effects are the same and it is treated as an exception in Alpha.

System-defined exceptions stem from the operating system's fundamental behavior and can be classified into three groups, depending on which system feature is responsible for the exception. The system-defined exceptions are due to the fact that Alpha is a distributed operating system that supports time constraints and atomic transactions. Because of the distributed nature of Alpha, processing nodes may fail while threads span nodes. This means that it is both possible for an invocation to fail, and for detached (i.e., *orphaned*) computations to be created. Therefore, the system must provide a means of indicating the failure of operation invocations, as well as a means for detecting and eliminating orphan threads. In addition, it is necessary that the system detect and eliminate orphaned portions of threads in a timely fashion. The longer that an orphaned computation continues to execute, the more resources it wastes (because such a computation is not able to complete, therefore the work it accomplishes is in vain), and the greater the amount of effort that will have to be expended to compensate for the actions it has performed. This means that an asynchronous thread section abortion mechanism is called for, to eliminate portions of threads regardless of whether they are blocked waiting on a remote invoke or are actively computing (i.e., either the head segment or body segments). This asynchronous thread abort mechanism must accomplish the orderly repair of "broken" threads (i.e., notify the new head of the thread when the orphaned section(s) have been successfully eliminated), and it must also be compatible with time-driven resource management objectives and mechanisms.

Any operating system that purports to support the time-driven management of resources must provide a means of dealing with the exceptions which stem from the inability of a computation to meet its given execution time constraints. The exceptions which occur when a time constraint expires require the same prompt attention as is needed for orphaned thread detection elimination. By definition, the continued execution of a computation that has missed a deadline (i.e., a hard time constraint) can no longer provide value to the system (and may in fact be counter-productive to the system). Therefore, an asynchronous exception mechanism is called for in this case as well. The system should provide a means of diverting control from a thread's normal flow of execution to an exception handler whenever a time limit (as defined by a thread's timeliness attributes) expires. Furthermore, this exception handling mechanism should preserve the state of the threads that miss time constraints, in order to permit the graceful recovery of a computation following an expired time constraint. As with all mechanisms in Alpha, this exception mechanism must work in concert with the system's overall time-driven resource management philosophy. The use of atomic transactions also imposes a requirement that the system support the (both synchronous and) asynchronous diversion of a thread's control that results when a transaction aborts (e.g., because of node failures or as a result of an explicit transaction abort command). As far as mechanisms for manipulating threads as a result of exceptions are concerned, the needs of atomic transactions are very similar to those of handling thread breaks or missed deadlines.

Machine-defined exceptions stem from events in the underlying hardware (e.g., divide by zero, bus error, non-existent memory, and access violations). All operating systems must deal with the various, machine-dependent ways that these exceptions are presented to the client. In general, these exceptions require that the computation that is responsible for the exception event be diverted from its normal path of execution, and vectored to some exception handling procedure.

When an exception occurs in Alpha, the objects affected are cleaned up (i.e., made consistent) by executing each of the abort block handler code sections from the head of the thread, back up stream until the proper level has been reached. This allows a thread to be cleaned up from within (by the thread itself, executing with its proper attributes), and this activity can span multiple objects and nodes as the thread works its way back upstream.

The asynchronous diversion of a thread's flow of control can leave an object in an inconsistent state. The system provides its own exception handling code to perform the cleanup necessary to ensure that system data structures are left in a consistent state following an exception (and that system resources are not "lost"). In addition to the system's exception handling code, the client can also provide application-specific exception handlers to perform specialized recovery/compensation actions, that can restore the application to a consistent state in a much more efficient manner than can be accomplished by the brute-force techniques that must be used by the system. The exception handling code that is associated with each exception block can be used to restore the consistency of the object's state, compensate for the effects of the aborted thread, or salvage usable results from the partially completed computation.

6.3.2 Optimizing for Exceptions

The robustness of Alpha is enhanced by optimizing the design and implementation for the exception cases, instead of the expected ones. Examples of the application of this principle can be found in the system's exception handling mechanisms, communications protocols, and operation invocation facility. The Alpha exception handling mechanism provides a unified mechanism for the management of exceptions associated with time constraints, atomic transactions and machine exceptions. For this mechanism, the normal condition is that the transaction commits, time constraint is satisfied, or no machine exception occurs; the exception case is that the transaction aborts, time constraint is not met, or a machine exception occurs. Performance is optimized in the exception case by trapping into the kernel on every exception block entry, in order to deposit the state information needed in case an exception occurs while executing within the block.

Another example is found in the remote operation invocation protocol that is used to provide the maintenance of threads as they extend across nodes. The normal case for this protocol is that the continuity of the thread is not broken by failures in nodes (or the communications network); the exception case is when a node (or the communications network) fails and the thread is broken. This mechanism's performance is optimized for the exception case (at the expense of the normal case) by the periodic exchange of keep-alive messages among the nodes that a thread extends across, rather than using only end-to-end time-outs on the remote invocations.

Furthermore, the kernel does not make use of hints in any form. In particular, the internal global identifiers used to access programming entities (e.g., threads and objects) in

Alpha does not include a (explicit) reference to the physical location of the intended entity. The expected case for this function would be that the hint is correct, while the exception case would be that the hint was wrong. The optimization of the exception case's performance is achieved by performing a multicast-like message transmission on each remote invoke, in order to locate the (dynamically relocatable) entities on a per-invocation basis. This is done instead of using a hint about the entity's physical location to perform a point-to-point remote procedure call, with a special sub-protocol that is to be performed to locate the target entity when the hint is wrong. While hints might speed references in a static system, when drastic dynamic reconfiguration of the system is underway, the use of hints incurs a higher than normal cost when the system can least afford it—i.e., in an exception condition.

6.4 Adaptability

One of the reasons for choosing the form of distributed computer system upon which the Alpha operating system is to execute has to do with the high degree of extensibility that is inherent in loosely coupled, bus-structured distributed systems.

The adaptability of the system resource management facilities in Alpha is supported to a great extent through the use of (both static and dynamically applied) application-specific information. In particular, a wide range of application- and system-specific attributes can be associated with computations and carried along with threads as they execute within the system. This application-specified information allows the system's resource management algorithms to adapt to the changing demands placed on the system by the application as the availability of system resources change.

In addition, the adaptability requirements of Alpha are addressed in two major ways—modularity for the operating system's clients is supported through the use of the object-oriented programming model supported by the operating system, and adaptability within the operating system itself is provided by a policy/mechanism separation approach [Hansen 70].

6.4.1 Object-Oriented Programming

The kernel provides a simple and uniform interface to its clients that centers around the operation invocation facility. The object programming abstraction supported by the operating system exhibits the same benefits associated with object-oriented programming abstractions in general, among which are information hiding, increased modularity, enhanced uniformity and simplicity of the programming interface, and reduced life-cycle costs [Bayer 79, Cox 86].

For reasons of adaptability, the attributes of threads (such as time constraints) are (by convention) modified in a block-structured and strictly nested within the operations of objects (e.g., a time constraint block begins and ends within the same object operation). This type of constraint provides modular, structured way of managing thread attributes, and is not unlike the way in which monitors confine all of the P and V operations on semaphores into a common module in order to add structure to, and avoid the problems of, distributed synchronization.

The well-defined interfaces defined by objects not only support information hiding by separating their interface specification from their internal implementation, but they also

permit all system services, including devices (as well as special hardware augmentations), to be presented to the programmer as objects. This feature allows the system to be modified by substituting, interposing, or overlaying modified objects, without affecting existing code. Furthermore, Alpha supports the simple and effective downward migration of functionality by allowing system functionality to be developed as client objects (for reasons of convenience) and then migrated into the kernel (for reasons of performance).

6.4.2 Policy/Mechanism Separation

The concept of policy/mechanism separation has been shown to be valuable in the design of modular operating system facilities [Hansen 71, Levin 75, Habermann 76]. Briefly, a policy is defined as a specification of the manner in which a set of resources are managed, and a mechanism is defined as the means by which policies are implemented [Hansen 70]. Policy/mechanism separation is a structuring methodology that involves the segregation of entities that dictate resource management strategies from entities that implement the low-level tactics of resource management.

Alpha's kernel is implemented as a collection of mechanisms from which policy decisions were carefully excluded. Each major logical function in the kernel is manifest in an individual mechanism, and a great effort was made to ensure a proper separation of concerns among these mechanisms. If the mechanisms are in fact pure (i.e., devoid of policy decisions) and complete, then it is possible to use them in implementing a wide range of system- and application-level facilities, and indeed entirely different operating systems. Adaptability is achieved through the separation of functions into mechanisms; implementation changes are restricted to individual mechanisms, and changes in system policy do not require changes in the functionality of mechanisms, just changes in the use of mechanisms.

The Alpha operating system supports policy/mechanism separation which allows the easy addition or modification of resource management policies—most of the major subsystems consist primarily of frameworks into which a wide range of differing policies can be inserted. Furthermore, the mechanisms that provide the Alpha programming model allow specialized, application-specific policies to be developed, either at system-build time or at run-time. The system does not enforce a particular policy on the user, but rather provides mechanisms that allow a wide range of policies to be implemented (e.g., in the area of concurrency control and exception handling).

An example of where the effects of policy/mechanism separation within the system are visible at the programming interface level can be seen in the kernel's mechanisms for managing object creation. The kernel mechanism that permits the creation of new objects does not include a policy for the placement of the newly created object. The kernel creates a new object local to the node at which the creation operation was invoked, instead of choosing a "desirable" location for the new object (e.g., on the least loaded node, on the node with the greatest amount of common data, or on the node nearest the needed I/O devices). In this way, the kernel's clients are free to apply the appropriate, possibly application-specific, policy to the initial placement of objects. This applies as well to the creation of replicated objects and the initial placement of the individual replicas. Furthermore, the kernel's native, flat object space allows arbitrary constraints on object relationships to be applied at the higher levels, to achieve the desired form of structure among objects, without suffering from potentially conflicting, system-imposed structures.

7 Acknowledgments

Many people contributed to this effort, and many others continue to contribute as the Archons project enters its next phase.

Doug Jensen is the founder of the Archons project, he obtained the support for the project over its 10-year existence, and provided most of the fundamental philosophy and concepts in Alpha. Doug continues to provide support and guidance for this research effort and an enhanced Release 2 of the Alpha operating system that is being developed at Kendall Square Research Corporation.

In his brief visit with the Archons project, Martin McKendry initiated the implementation effort that has become Alpha and provided many of the initial implementation concepts. Ray Clark worked on defining and implementing the basic system abstractions and is now performing his thesis research in the area of real-time transactions within the Alpha context.

Sam Shipman and worked on various subsystems and support tools, and worked on refining and completing the system design and implementation. David Maynard also worked on various aspects of the system's design and implementation, and acted as the project liaison with General Dynamics during our joint C² demonstration effort. David is now working on a thesis in the area of real-time scheduling for decentralized computers with multiprocessor nodes. Huay-Yong Wang worked on the scheduling subsystem and the design and implementation of various other system functions.

Other project members that contributed to the project are Jeff Trull, Chuck Kollar, Bruce Taylor, Don Lindsay, and Dan Reiner.

Thanks is due to Tom Lawrence and Dick Metzger, the Archons project's prime sponsors at the Rome Air Development Center. Additionally, we would like to thank Russell Kegley and Calvin Head of the Fort Worth Division of General Dynamics Corporation for their assistance in our joint C² application development effort.

UNIX is a trademark of AT&T Bell Laboratories.

VAX and VMS are trademarks of Digital Equipment Corporation.

Ethernet is a trademark of Xerox Corporation.

Ada is a trademark of the United States Department of Defense.

Accent is a trademark of Perq Corporation.

VRTX is a trademark of Hunter & Ready Corporation.

RTU is a trademark of Massachusetts Computer Corporation.

References

- [Ada 83] United States Department of Defense.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815A-1983.
Springer-Verlag, New York, 1983.
- [Anderson 81] Anderson, T. and Lee, P. A.
Fault Tolerance: Principles and Practice.
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
- [Accetta 86] Accetta, M. J., Baron, R. X., Golub, D., Rashid, R. F., Tevanian, A.
and Young, M.
A New Kernel Foundation for UNIX Development.
In *Proceedings of the Summer 1986 USENIX Technical Conference
and Exhibition*, pages 35-41, June, 1986.
- [Avizienis 78] Avizienis, A.
Fault-Tolerance: The Survival Attribute of Digital Systems.
Proceedings of the IEEE 66(10):1109-1125, October, 1978.
- [Bayer 79] Bayer, R., Graham, R. M. and Seegmueller, G. (editors).
*Lecture Notes in Computer Science. Volume 60: Operating Systems:
An Advanced Course.*
Springer-Verlag, Berlin, West Germany, 1979.
- [Boebert 78] Boebert, W. E.
Concepts and Facilities of the HXDP Executive.
Technical Report 78SRC21, Honeywell Systems & Research Center,
March, 1978.
- [Boehm 81] Boehm, B. W.
Advances in Computer Science and Technology: Software Engineer-
ing Economics.
Prentice Hall, Englewood Cliffs, New Jersey, 1981.
- [Cheriton 84] Cheriton, D. R.
The V Kernel: A Software Base for Distributed Systems.
IEEE Software 1(2):19-43, January, 1984.
- [Clark 83] Clark, R. K. and Shipman, S. E.
The Archons Testbed—A Requirements Study.
Archons Project Technical Report #83051, Department of Computer
Science, Carnegie-Mellon University, May, 1983.
- [Clark 88] Clark, R. K.
Operating System Kernel Support for Real-Time Atomic Transactions.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon
University.
In progress.

- [Cox 86] Cox, B. J.
Object-Oriented Programming.
Addison-Wesley, Reading, Massachusetts, 1986.
- [DRC 86] Dynamics Research Corporation
Distributed Systems Technology Assessment for SDI.
Technical Report E-12256U, Electronic Systems Division, USAF
Systems Command, September, 1986.
- [Farber 72] Farber, D. J. and Larson, K. C.
The System Architecture of the Distributed Computer System—The
Communications System.
In *Proceedings, Symposium on Computer-Communications Networks
and Teletraffic*, pages 21-27. Polytechnic Institute of Brooklyn,
April, 1972.
- [Fitzgerald 85] Fitzgerald, R. and Rashid, R. F.
The Integration of Virtual Memory Management and Interprocess
Communication in Accent.
In *Proceedings, Tenth Symposium on Operating System Principles*,
pages 13-14. ACM, November, 1985.
- [Franta 81] Franta, W. R., Jensen, E. D., Kain, R. Y. and Marshall G. D.
Real-Time Distributed Computer Systems.
Advances in Computers 20:39-82, 1981
- [GD 80] General Dynamics
*Computer Program Product Specification for the System Function
Processor Operational Flight Program for the F-16 Multination
Staged Improvement Program, Block 30*.
Technical Report #CPCI 7175-1A00, General Dynamics Corporation,
December, 1980.
- [Gifford 79] Gifford, D. K.
Weighted Voting for Replicated Data.
In *Proceedings, Seventh Symposium on Operating Systems Principles*,
pages 150-162. ACM, December, 1979.
- [Glass 80] Glass, R. L.
Real-Time: The "Lost World" of Software Debugging and Testing.
Communications of the ACM 23(5):264-271, May, 1980.
- [Goldberg 83] Goldberg, A. and Robson, D.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, Massachusetts, 1983.
- [Habermann 76] Habermann, A. N., Flon, L. and Coopride, L.
Modularization and Hierarchy in a Family of Operating Systems.
Communications of the ACM 19(5):266-272, May, 1976.

- [Hansen 70] Brinch Hansen, P.
The Nucleus of a Multiprogramming System.
Communications of the ACM 13(4):238-250, April, 1970.
- [Hansen 71] Brinch Hansen, P.
RC4000 Software Multiprogramming System.
A/C Regnecentralen, Copenhagen, 1971.
- [Herlihy 86] Herlihy, M. P.
Using Type Information to Enhance the Availability of Partitioned Data.
Technical Report CMU-CS-85-119, Department of Computer Science, Carnegie-Mellon University, April, 1985.
- [Henize 84] Henize, J. A.
Understanding Real-Time UNIX.
MASSCOMP Technical Report, January, 1986.
- [Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June, 1975.
- [Jensen 76a] Jensen, E. D.
The Implications of Physical Dispersal on Operating Systems.
Workshop on Distributed Processing, Brown University, Providence, Rhode Island, August, 1976.
- [Jensen 76b] Jensen, E. D. and Anderson, G. A.
Feasibility Demonstration of Distributed Processing for Small Ships Command and Control Systems.
Final Report N00123-74-C-0891, Honeywell Systems & Research Center, August, 1976.
- [Jensen 78a] Jensen, E. D.
The Honeywell Experimental Distributed Processor—An Overview.
Computer 11(1):137-147, January, 1978.
- [Jensen 78b] Jensen, E. D., Marshall, G. D., White, J. A. and Helmbrecht, W. F.
The Impact of Wideband Multiplex Concepts on Microprocessor-Based Avionic System Architectures.
Technical Report AFAL-TR-78-4, Honeywell Systems & Research Center, February, 1978.
- [Jensen 84] Jensen, E. D. and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System.
In *Distributed Processing Technical Committee Newsletter*. IEEE, June 1984.

- [Jones 79] Jones, A., Chansler, R., Durham, I., Schwans, K. and Vegdahl, S.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings, Seventh Symposium on Operating System Principles*, pages 117-127. ACM, December, 1979.
- [Lampson 81] Lampson, B. W., Paul, M. and Siebert, H. J. (editors).
Lecture Notes in Computer Science. Volume 105: *Distributed Systems—Architecture and Implementation*.
Springer-Verlag, Berlin, 1981.
- [Lechowski 86] Lechowski, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review 14(1):44-53, May 1986.
- [Lehman 85] Lehman, M. M. and Belady, L. A.
Program Evolution: Processes of Software Change.
Academic Press, London, 1985.
- [Leinbaugh 80] Leinbaugh, D. W.
Guaranteed Response Times in a Hard-Real-Time Environment.
IEEE Transactions on Software Engineering SE-6(1):85-91, January, 1980.
- [Levin 75] Levin, R., Cohen, E., Corwin, W., Pollock, F., and Wulf, W.
Policy/Mechanism Separation in Hydra.
In *Proceedings, Fifth Symposium on Operating Systems Principles*, pages 132-140. ACM, November, 1975.
- [Liskov 84] Liskov, B. H.
Overview of the Argus Language and System.
Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February, 1984.
- [Liskov 85] Liskov, B. H., Herlihy, M. P. and Gilbert, L.
Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing.
Technical Report CMU-CS-85-168, Department of Computer Science, Carnegie-Mellon University, October, 1985.
- [Liu 73] Liu, C. L. and Leyland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
Journal of the ACM 20(1):46-61, 1973.
- [Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May, 1986.

-
- [Maynard 88] Maynard, D. P., Clark, R. K., Kegley, R. B., Keleher, Northcutt, J. D., Shipman, S. E., and Zimmerman, B. A.
An Example Real-Time Command and Control Application.
Archons Project Technical Report #88032, Department of Computer Science, Carnegie-Mellon University, March, 1988.
- [McQuillan 80] McQuillan, J. M., Richer, I. and Rosen, E. C.
The New Routing Algorithm for the ARPANET.
IEEE Transactions on Communications COM-28(5):711-719, May, 1980
- [Metcalf 72] Metcalf, R. M.
Strategies for Interprocess Communication in a Distributed Computing System.
In *Proceedings, Symposium on Computer-Communications Network and Teletraffic*, pages 519-526. Polytechnic Institute of Brooklyn, April, 1972.
- [Mockapetris 77] Mockapetris, P. V., Lyle, M. and Farber, D. J.
On the Design of a Local Network Interface.
In *Proceedings, Information Processing 77*. IFIP, 1977.
- [Moss 85] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
The MIT Press, Cambridge, Massachusetts, 1985.
- [Nelson 81] Nelson, B. J.
Remote Procedure Call.
Ph.D. Thesis. Department of Computer Science, Carnegie-Mellon University, May, 1981.
- [Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.
- [Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Programming Model
Archons Project Technical Report #88021, Department of Computer Science, Carnegie-Mellon University, February, 1988.
- [Northcutt 88b] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.
- [Ousterhout 80] Ousterhout, J. K., Scelza, D. A. and Sindhu, P. S.
Medusa: An Experiment in Distributed Operating System Structure (Summary).
Communications of the ACM 23(2):92-105, February, 1980.
-

- [Parnas 77] Parnas, D. L.
Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems.
Technical Report 8047, Naval Research Laboratory, December, 1977.
- [Quirk 85] Quirk, A. B.
Verification and Validation of Real-Time Software.
Springer-Verlag, Berlin, 1985.
- [Randell 78] Randell, B., Lee, P. A. and Treleaven, P. C.
Reliability Issues in Computing System Design.
Computing Surveys 10(2):123-165, June, 1978.
- [Rashid 81] Rashid, R. F. and Robertson, G. G.
Accent: A Communications Oriented Network Operating System Kernel.
Technical Report CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University, April 1981.
- [Ready 86] Ready, J. F.
VRTX: A Real-Time Operating System for Embedded Microprocessor Applications.
IEEE Micro 6(4):8-17, August, 1986.
- [Savitzky 85] Savitzky, S. R.
Real-Time Microprocessor Systems.
Van Nostrand Reinhold, New York, 1985.
- [Sha 85] Sha, L.
Modular Concurrency Control and Failure Recovery—Consistency, Correctness and Optimality.
Ph.D. Thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1985.
- [Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, April, 1988.
- [Stankovik 88] Stankovik, J. A.
Real-Time Computing Systems: The Next Generation.
Technical Report COINS 88-06, University of Massachusetts, January 1988.
- [Smith 79] Smith, R. G.
The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.
In *Proceedings, First International Conference on Distributed Computing*, pages 185-192. IEEE, October, 1979.

-
- [Thomas 78] Thomas, R. H., Schantz, R. E. and Forsdick, H. C.
Network Operating Systems.
Technical Report 3796, Bolt, Beranek and Newman, 1978.
- [Thompson 80] Thompson, J. R., Ruspini, E. H. and Montgomery, C. A.
TAC C³ Distributed Operating System Study.
Technical Report RADC-TR-79-360, Operating Systems, Inc. for
Rome Air Development Center, January, 1980.
- [Wallis 84] Wallis, D. E.
DSDB/DTN Machine Level Architecture and Principles of Operation.
Internal Technical Report IBM-2849070, May, 1984.
- [Wirth 77] Wirth, N.
Towards a Discipline of Real-Time Programming.
Communications of the ACM 20(8):557-585, August, 1977.
- [Wittie 79] Wittie, L. D.
A Distributed Operating System for a Reconfigurable Network Computer.
In *Proceedings, First International Conference on Distributed Computing Systems*, pages 669-677. IEEE, October, 1979.
- [Wulf 81] Wulf, W. A., Levin, R. and Harbison, S. P.
Hydra/C.mmp: An Experimental Computer System.
McGraw/Hill, New York, 1981.
-

Table of Contents

Abstract.....	C-1
1 Introduction.....	C-2
1.1 The Alpha Application Effort.....	C-2
1.2 Application Selection.....	C-3
1.3 Application Development.....	C-4
1.4 Caveats.....	C-5
1.5 Report Outline.....	C-5
2 The Alpha Operating System.....	C-6
2.1 Alpha Programming Model.....	C-6
2.2 Time-Driven Resource Management.....	C-8
2.2.1 Time-Value Functions.....	C-8
2.2.2 Best-Effort Resource Management.....	C-9
2.3 Support for Survivability.....	C-9
2.4 Pre-Release Alpha Version 0.5.....	C-11
2.5 The Alpha Distributed Computer System Testbed.....	C-11
2.5.1 Distributed System Testbed.....	C-12
2.5.2 Development and Control System.....	C-13
3 Application Requirements.....	C-14
3.1 Scenario Description.....	C-14
3.2 Application Processing Requirements.....	C-15
3.2.1 Tracking.....	C-16
3.2.2 Mission Planning.....	C-17
3.2.3 Weapon Systems Control.....	C-17
3.3 Application Timeliness Requirements.....	C-18
3.3.1 Radar Plot Correlation.....	C-18
3.3.2 Weapons Guidance.....	C-18
3.3.3 SAM Control.....	C-19
3.4 Application Survivability Requirements.....	C-19
4 Experimental Environment.....	C-21
4.1 Scenario Simulator.....	C-21
4.2 Operator Console.....	C-23
4.3 Experimental Control Console.....	C-25
4.4 Communication Interface.....	C-27
5 Air Defense System Design.....	C-28
5.1 Design Approach.....	C-28
5.2 Functional Decomposition.....	C-29
6 Air Defense System Objects.....	C-31
6.1 Tracking Objects.....	C-32
6.1.1 Plot Correlator.....	C-32

6.1.2	Track Database.....	C-33
6.2	Mission Planning Objects	C-34
6.2.1	Track Handler	C-34
6.2.2	Track Identifier	C-35
6.2.3	Threat Assessment	C-35
6.2.4	Weapons Manager	C-36
6.3	Weapon Systems Control Objects	C-37
6.3.1	Interceptor Controller.....	C-38
6.3.2	Missile Controller	C-39
6.3.3	AWACS Controller.....	C-40
6.3.4	SAM Controller	C-41
6.3.5	SAM Engagement.....	C-41
6.4	System Support Objects	C-42
6.4.1	Communication Interface.....	C-42
6.4.2	Serial I/O	C-43
6.4.3	Network I/O	C-43
6.4.4	Node Status	C-44
6.4.5	Distributed Services Manager.....	C-45
6.4.6	Local Services Manager.....	C-46
6.4.7	Recovery Manager	C-46
7	Air Defense System Threads	C-47
7.1	Tracking Threads	C-48
7.1.1	Plot Correlation	C-48
7.1.2	Track Database Maintenance	C-48
7.2	Mission Planning Threads	C-48
7.2.1	New Track Dispatching	C-48
7.2.2	Track Processing	C-48
7.3	Weapon Systems Control Threads.....	C-49
7.3.1	Interceptor Launch and Control	C-49
7.3.2	Missile Launch and Control.....	C-49
7.3.3	AWACS Launch and Control	C-49
7.3.4	SAM Launch and Control	C-49
7.3.5	SAM Monitoring.....	C-49
7.4	System Support Threads	C-50
7.4.1	Input Processing.....	C-50
7.4.2	Node Status Monitoring	C-50
7.4.3	Node Lifeline	C-50
7.4.4	Node Recovery Processing	C-50
8	Technology Evaluation	C-51
8.1	Distribution and Survivability	C-51
8.1.1	Object Distribution	C-52
8.1.2	Thread Distribution.....	C-53
8.1.3	Node Failure Response	C-54
8.2	Time-Driven Resource Management.....	C-55
8.2.1	Application Time-Value Functions.....	C-56

8.2.2	System Overload Behavior	C-58
9	Conclusions	C-60
9.1	Lessons Learned.....	C-60
9.1.1	Decentralized Computing	C-60
9.1.2	Programming Model	C-60
9.1.3	Time-Driven Resource Management.....	C-60
9.1.4	System Survivability	C-61
9.1.5	Program Management	C-61
9.2	Future Directions	C-62
9.2.1	Related Research.....	C-62
9.2.2	System Enhancements	C-62
9.2.3	Future Applications.....	C-63
	References	C-64
	Appendix I: Application Program Example.....	C-66

List of Figures

Figure 1	Example Object	C-7
Figure 2	Example Time-Value Functions	C-8
Figure 3	Alpha Distributed Computer System Testbed Structure	C-12
Figure 4	Alpha Testbed Node Architecture	C-13
Figure 5	High-Level System Functions	C-16
Figure 6	Overall System Architecture	C-22
Figure 7	Operator Console Display	C-24
Figure 8	Experimental Control Console Display	C-25
Figure 9	Node Information Display	C-26
Figure 10	Air Defense System Object Types.....	C-31
Figure 11	Air Defense System Threads	C-47
Figure 12	Typical Distribution of Objects and Threads	C-51
Figure 13	Object Distribution Classifications.....	C-52
Figure 14	Reconfiguration After a Node Failure	C-54
Figure 15	ADSP Time-Value Function Examples.....	C-56
Figure 16	Normal Behavior	C-58
Figure 17	Overload Behavior.....	C-59

Abstract

This report describes the design, implementation, and evaluation of a demonstration command, control and battle management (C²/BM) system developed jointly by Archons project researchers at Carnegie Mellon University (CMU) and by technical staff members at General Dynamics Corporation (GD). Over the course of a few months, this team defined a realistic air defense scenario, designed a distributed, real-time C²/BM system for this scenario, and developed a prototype implementation of the design using an early version of the Alpha operating system. In addition, a support environment was constructed to exercise the application. The primary goals of the application effort were to demonstrate the decentralized, real-time technology incorporated in the Alpha operating system and to evaluate that technology in a realistic context. To that end, the application exercises the distribution, real-time, and survivability mechanisms provided by Alpha.

By transferring Alpha technology from a research environment into an industrial setting, Archons researchers were able to obtain feedback from professional designers and system builders. This feedback validated the belief that the programming model and key support mechanisms provided by Alpha are well-suited to the design and construction of large, distributed, real-time control applications.

1 Introduction

This report describes the design, implementation, and evaluation of a real-time command, control, and battle management (C²/BM) application developed to run on an early version of the Alpha operating system. This was an experimental effort aimed at investigating the efficacy of using Alpha technology as a basis for developing large, distributed, real-time systems. Over the course of a few months, a small team of people from the Archons project at Carnegie Mellon University (CMU) and from the Fort Worth Division of General Dynamics Corporation (GD) designed and developed both an air defense system prototype (ADSP) and a support environment in which to exercise it. Using this experimental prototype, we have demonstrated that Alpha's basic abstractions and resource management techniques can be used in the development of the complex real-time systems that will be required in the future.

Alpha is an adaptable, decentralized operating system for real-time applications. It is being developed as part of the Archons project's on-going research into distributed real-time systems. Alpha is a new kind of operating system that is unique in three significant ways. First, Alpha is decentralized, providing reliable resource management transparently across physically dispersed nodes. This allows distributed application programming to be done as though it were centralized. Second, Alpha provides comprehensive, high-technology support for real-time systems. In particular, it supports supervisory control applications (*e.g.*, industrial automation, combat platform management) which are characterized by predominately aperiodic activities that have critical time constraints (such as deadlines). Third, Alpha provides a unique combination of mechanisms for supporting the construction of highly survivable applications. Chapter 2 provides a brief introduction to Alpha. A book [Northcutt 87] and a series of technical reports (listed in the references) describe Alpha in more detail.

1.1 The Alpha Application Effort

Early in 1987, the Alpha research group at CMU began an effort aimed at evaluating and demonstrating the potential of Alpha technology in the context of a realistic, distributed, real-time application. Although Alpha implementation was still in its early stages (Pre-Release 0.2), we felt that building a realistic application would provide useful feedback so we could evaluate and, if necessary, redirect our research and development efforts.

We use the term "realistic," as opposed to "real," because experimenting with real applications of the kind that Alpha is designed to support would be difficult and expensive. Such applications are typically embedded—controlling vehicles, factories, or other complex machinery that interacts with the physical world. Real applications are also very complex. This complexity often impedes experimentation by making it difficult to isolate individual effects so that the results of various changes can be observed. To overcome these problems, we decided to build an experimental prototype containing the important components of a real application, but with few of the extra features that might limit the visibility of critical effects.

Our specific objective was to create an application that was reasonably representative of supervisory-level, real-time C²/BM systems. It was important that the application

have sufficiently complex and stringent requirements to benefit from Alpha's support for distributed processing, time-driven scheduling, and system survivability. Some of the specific requirements were:

- an application that could take advantage of the potential performance and survivability benefits offered by Alpha's transparent distributed processing,
- an application that could demonstrate system survivability and graceful degradation through automatic reconfiguration after node failures and through (re)integration of new or repaired nodes,
- an application that could demonstrate the scheduler's ability to ensure that: 1) when possible, application time constraints are satisfied, and 2) when overloaded, application-specified policies are used to control the allocation of the limited resources available,
- an application with dynamically varying time constraints that could exercise the flexibility and power of the best-effort scheduling techniques.

Finally, we required an application that could graphically illustrate these capabilities during a fairly short presentation that involved running the application software while explaining its behavior to an audience in real time. This final requirement allowed us to demonstrate our results to our research sponsors and other interested parties.

Since we needed a realistic application, we decided to team with an industrial partner. This partner would provide expertise in real-time command and control applications and share the effort of implementation. The Alpha group would gain insight into the requirements of the application and of the application programmers (in terms of development and debugging tools). In return, the industrial partner would benefit from the transfer of technology from the Archons project. Both groups would gain valuable experience in developing decentralized real-time systems.

1.2 Application Selection

Bearing in mind the objectives stated in Section 1.1, the Alpha group began the search for a suitable application and partner. We considered a wide range of application types—vehicle control (ship, airplane, spacecraft), process control (factory automation), command and control (air traffic control), and spoke with several organizations (e.g., General Dynamics, Jet Propulsion Laboratory, IBM).

In March 1987, a group at General Dynamics approached the Archons project and indicated their interest in advanced technology for future C²/BM systems. We told them of our interest in an application for Alpha, and together settled on the idea of building a prototype for a coastal air defense system inspired by the requirements of the Air Defense Initiative (ADI)—a U.S. Air Force defense program designed to protect against airborne attackers of the 1990's and beyond.

In the chosen scenario (described in detail in Chapter 3), the system coordinates the defense of the northeastern United States against airborne threats approaching over the Atlantic Ocean. The major functions of the system are to locate, identify, track and intercept threats before they inflict damage. The primary threats are manned bombers, manned cruise missile carriers, cruise missiles, and jamming devices. Available defensive assets include over-the-horizon (OTH) ground-based radars, airborne early warn-

ing aircraft, manned interceptors, medium-range anti-missile missiles, and point-defense surface to air missiles (SAMs). The system uses these assets to gather data from sensors, track the targets, assess their threat potential, launch weapons, and control weapon flight paths until the threats are destroyed. The system achieves its goal if no bombers or cruise missiles reach their targets.

1.3 Application Development

After the Archons project and General Dynamics agreed to cooperate on building an application for Alpha, a technology development and exchange effort was initiated. The effort was carried out from April to November 1987, with approximately six months of this time being spent on the actual system implementation.

General Dynamics brought to the effort their knowledge of real-time C²/BM applications, their implementation experience with air defense applications, their experience with environment simulators and scenarios, their expertise in operator console and user interface design, and their skills in producing software to meet industrial quality standards. The Alpha group contributed their expertise in designing distributed real-time systems and the Archons project's advanced technology for decentralized real-time operating systems embodied in Alpha.

Working together, personnel from the Alpha group and from General Dynamics designed the scenario and specified the functions of the application system. This cooperation ensured that the application scenario and implementation were reasonably representative of real-world systems, and that the system would demonstrate some of the most important features of Alpha.

General Dynamics dedicated three full-time employees to the task of application design and implementation during the period of the joint effort. General Dynamics implemented the application system running under Alpha. They also designed and implemented the experimental support environment running under UNIX. The support environment (described in Chapter 4) includes the Scenario Simulator, the Experimental Control Console, the Operator Console, and the Communication Interface.

At CMU, four people worked full-time on the design and implementation of Alpha. CMU provided GD with pre-release software, culminating in Alpha 0.5. They provided advice on the proper use of the Alpha object-oriented programming model and other Alpha facilities, and assisted in implementing some portions of the application code. CMU personnel also assembled a duplicate of the Archons distributed systems testbed (See Section 2.5) at General Dynamics.

The Alpha group dedicated a fifth person to act as coordinator and liaison. The liaison spent approximately half-time working in Pittsburgh at CMU and half-time in Fort Worth at GD for the duration of the project (travelling between cities at two-week intervals). Work proceeded in a loosely coupled fashion, with the Alpha group at CMU concentrating on the operating system, and the application group at General Dynamics concentrating on the ADSP and its support environment.

The defense system prototype and its support environment was initially demonstrated to our research sponsors in December 1987. In March 1988 it was shown to our spon-

sors, DoD representatives, members of industrial technical staffs, and academic researchers. Both demonstrations were well received.

1.4 Caveats

It is worth repeating that the work described in this report was directed toward a demonstration that previews technology still under development. The application is realistic in that it is representative of the kinds of processing that take place in real-world systems. Although inspired by the ADI environment, the threat, doctrine, and tactics in the application are purely notional. Finally, the application software runs under pre-release Alpha 0.5. This early version of the operating system lacks several features that would further simplify the design and implementation of large distributed systems.

1.5 Report Outline

The remainder of this report describes the application in more detail and analyzes the results of the effort. Chapter 2 provides a system context by briefly describing the Alpha operating system and the Alpha distributed system testbed. Chapter 3 describes the application scenario and requirements in greater detail. Chapter 4 describes the experimental support environment in which the air defense system operates. Chapter 5 provides an introduction to the design of Alpha-based applications, and describes the high-level design of the ADSP. Chapter 6 describes each of the application object types, while Chapter 7 outlines the application threads. Chapter 8 explores the key technology issues of time-driven resource management and system survivability in the context of the ADSP. Finally, Chapter 9, evaluates our experience with the application and outlines ongoing Alpha research and development. An appendix at the end of the report provides a sample of the actual application code.

2 The Alpha Operating System

Alpha is an operating system designed to support complex, mission-oriented, real-time applications. These applications often have requirements that are very different from those usually considered by operating system designers. In particular, they require new types of support for distribution, real-time resource management, and system survivability and adaptability. Unlike other systems, Alpha has been crafted "from the ground up" to support the design and operation of such demanding applications in a reliable and maintainable manner.

Alpha is a *decentralized* operating system. It is oriented towards systems that may have anywhere from one to 100 physically dispersed processing nodes contributing to a single application or mission. By logically integrating these nodes into a single computer system, Alpha provides transparent access to distributed resources, and offers unified resource management (not just limited resource sharing) across node boundaries. By eliminating centralized facilities, Alpha allows applications to obtain the availability and survivability benefits of physical distribution without encountering the difficult problems associated with conventional distributed programming.

In addition to providing uniform resource management across node boundaries, Alpha supports *time-driven* management of vital resources such as processor cycles, communication access, and physical memory. Unlike static or synchronous scheduling techniques, Alpha's method of time-driven resource management supports supervisory-level real-time systems that may be dominated by dynamic and aperiodic activities with critical time constraints.

The prototype implementation (Release 1) of Alpha was created at CMU after more than ten years of experience with and research into the requirements of large, distributed, real-time systems. Further research and development continues at Concurrent Computer Corporation where Releases 2 and 3 of Alpha are being designed and implemented.

2.1 Alpha Programming Model

The Alpha programming model is based on the concepts of *objects* and *threads*. This model differs from process-based models by distinguishing data and executable code (objects) from the points of control that "animate" the code (threads). By making this distinction, Alpha is able to offer greater concurrency, better data encapsulation and more uniform resource management.

In Alpha, objects can contain both code and data. At the highest level of abstraction, objects correspond closely to abstract data types. Each object has a well-defined interface and is accessible only through the entry points or *operations* which it defines. Figure 1 shows a sample object with three operations (*Initialize*, *Insert*, and *Remove*) that define its external interface. Alpha enforces the object interface by placing every object in a separate, hardware-protected address space, and by controlling entry into objects with an *operation invocation* mechanism.

Unlike some object-oriented systems, objects in Alpha are passive. For computation to occur in an object, there must be one or more threads active in it. Alpha threads are the schedulable points of control that actually execute instructions. Intuitively, they are similar to "processes" in other systems. Each thread consists of an execution context

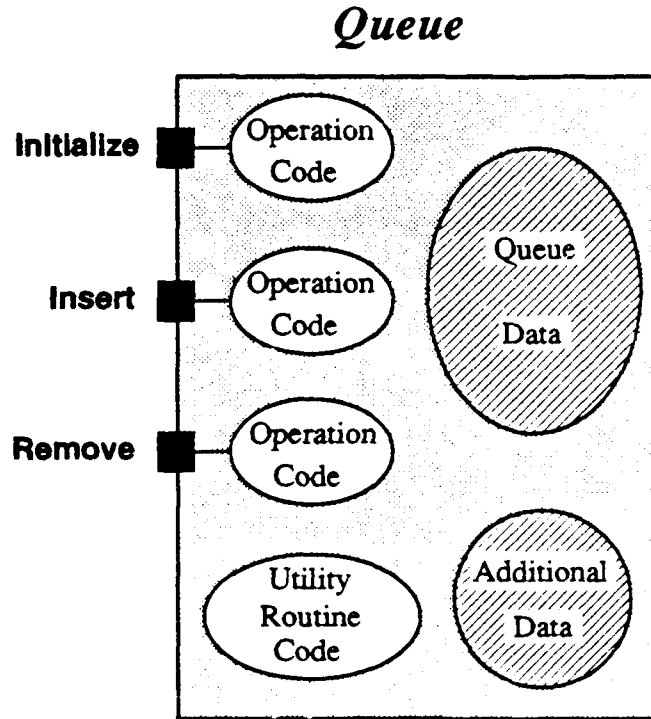


Figure 1: Example Object

that includes thread-specific data (*e.g.*, register values and stack information) and a set of attributes. The thread attributes include information, such as time constraints and reliability requirements, that is used by the system resource managers (*e.g.*, to ensure that deadlines are met).

When a thread is created, it begins executing at an application-selected operation in a specified object. Once running, it can travel to other objects by invoking operations on them. Alpha's threads are unique in that they are not constrained to remain on a single node. If an operation is invoked on an object at a different node, the thread transparently travels to the remote node, carrying its thread attributes with it. These attributes allow resources to be managed consistently throughout the entire decentralized computer.

Alpha does not artificially limit the level of concurrency within objects. At any given time, there may be zero, one, or several threads active in a particular object. The operating system provides concurrency control mechanisms such as locks and semaphores that the programmer can use to synchronize thread execution when required by the application.

Object-oriented programming offers many benefits including simplified design, improved modularity, and increased reusability of software components. Several books, such as [Cox 86], discuss the general merits of object-based systems. Alpha's object/thread model has specific advantages that make it well-suited for distributed, real-time systems. A technical report on the requirements and rationale for Alpha [Northcutt 88a] explains these advantages in greater detail.

2.2 Time-Driven Resource Management

One of Alpha's primary goals is to support highly complex and dynamic real-time applications. A major way that Alpha supports such applications is by resolving contention for system resources in a global, dynamic, and (most importantly) time-driven manner. Large real-time systems often consist of several concurrent activities, each of which may have critical timeliness requirements. In Alpha, these activities are embodied as threads. The thread abstraction not only serves as a means of capturing the notion of application activities in a system context, but provides a mechanism for notifying the operating system of any application-specific time constraints. Activities that have timeliness requirements specify those constraints dynamically as part of the attribute information that accompanies a thread.

2.2.1 Time-Value Functions

An essential prerequisite for being able to satisfy an application's timeliness requirements is providing a way for the system designer to express those requirements both concisely and accurately. In Alpha, time constraints are specified using the concept of *time-value functions*—functions that express the time-dependent value to the system of completing a specific phase of a computation [Jensen 75]. Time-value functions allow the designer to express requirements with much greater fidelity than the simple priorities or hard deadlines that other systems employ. Time-value functions are based on the idea that completing a computation has a certain value to the system. In real-time applications, this completion value often varies with time. For example, the classical concept of a hard deadline describes a task that has a constant positive completion value before its deadline, and zero value after the deadline (see Figure 2). Not all real-time activities have hard deadlines however. Other tasks may have critical times when their completion is *most* valuable, but may have non-zero completion value even after that time has passed. Such constraints are often known as "soft" time constraints since they do not have a single time, or hard deadline, when the completion value instantaneously drops to zero. It is also possible that some tasks are less valuable if completed too quickly (e.g., when their input data improves in quality as time progresses).

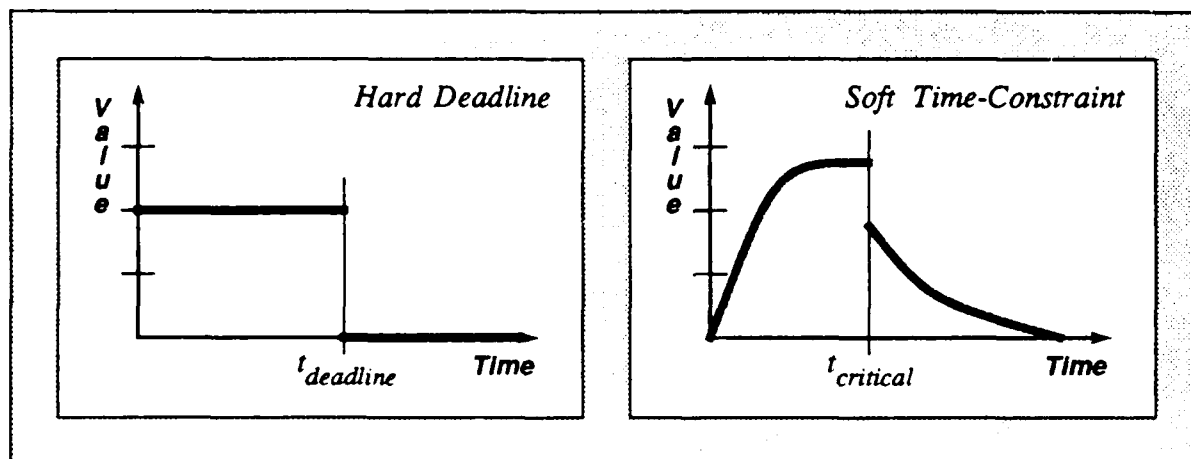


Figure 2: Example Time-Value Functions

Due to the dynamic nature of Alpha's intended applications, Alpha allows time constraints to be computed and specified at run-time. This allows applications to react to changes in the available resources or the environment. Several technical reports [Northcutt 88b] [Shipman 88] explain the features and usage of time-value functions more thoroughly.

2.2.2 Best-Effort Resource Management

Simply being able to express an application's timeliness requirements to the system would not be very useful if there were no way of utilizing the information to manage system resources. Alpha supports a wide range of scheduling policies that may use all, part, or none of this information. One particular class of policies is being developed as part of the Alpha group's research into advanced real-time systems. This class of policies uses a technique known as *best-effort* resource management to satisfy the application time constraints whenever possible, and to facilitate graceful degradation (as defined by the application) when resource demands exceed the available supply. Best-effort policies evaluate the time-value functions of all contending threads collectively. If sufficient resources are available, the threads are scheduled in order of their critical times (the time after which the value of completing the activity diminishes). If the system is overloaded, threads are (temporarily) discarded from the schedule until enough resources are available for the activities that are scheduled to satisfy their time constraints.

Several policies may be used for deciding which threads to remove. The load reduction policy used for the ADSP attempts to maximize the total *value* accrued to the system during overloads. The value of completing time-critical tasks is specified by the application designer by means of time-value functions. When resource demands are too great, the best-effort policies ensure that time and effort are spent on activities that are potentially the most valuable. Equally as important, the policies ensure that resources are not wasted on activities that have little or no chance of making a positive contribution to the application.

Alpha's mechanisms for time-driven resource management are described more thoroughly in many of the technical reports listed in the references. More details on the best-effort algorithms are available in [Locke 86], [Clark 88] and others.

2.3 Support for Survivability

The mission-oriented systems for which Alpha is designed often need to continue operating even if parts of the computer system fail. Alpha provides many facilities that support the construction of systems that can survive and adapt to node failures and other sudden or gradual changes in the environment. These facilities are complementary to, but independent of, hardware mechanisms that enhance the reliability or fault tolerance of the computer hardware. Alpha does not impose policies on how failures should be handled. Instead, the operating system provides mechanisms that system designers can use separately or in concert to construct a system that includes the performance vs. survivability trade-offs that are appropriate for the application. The survivability tools that Alpha provides include methods for containing the effects of faults, mechanisms for handling exceptions in a uniform manner, facilities for ensuring the availability and consistency of vital data and services, and mechanisms for ensuring graceful degradation and the timely distribution of resources under overload conditions.

When hardware or software faults occur, it is important to contain their effects insofar as possible. Alpha's object-based model limits fault propagation by placing each object in a separate (hardware-enforced) address space, and by separating software components into system-enforced protection domains. These protection domains are defined using a capability mechanism that controls interactions between objects. Threads are only allowed to invoke operations on objects for which they have explicit capabilities (which cannot be forged). This protection of data and control over thread propagation helps the system designer prevent faults from spreading unchecked throughout the system.

For detecting failures, Alpha provides a uniform exception handling mechanism that supports a wide range of user- and system-defined events. Because of the distributed nature of Alpha, processing nodes may fail while threads span nodes. This means both that invocations can fail because an object no longer exists and that portions of a computation can become detached (or *orphaned*) because a thread spans a node that fails. Alpha provides mechanisms for indicating the failure of operation invocations and for detecting and eliminating orphan threads. Alpha also supports the notion of time-constraint exceptions that occur when a computation cannot meet its timeliness requirements (e.g., because of a system overload). Since timeliness requirements are considered part of the correctness requirements for the system, such exceptions may be as vital as those caused by node failures. In Alpha, both types of exceptions are handled by the same mechanisms.

In order to respond effectively when a failure does occur, vital services must remain available and vital data must remain both available and (in some cases) consistent. The Alpha kernel has built-in functionality that supports object replication and atomic transaction mechanisms. By using *exclusive* replication (replication without data consistency between replicas), objects that provide vital services can be made available even if one or more of the replicas are destroyed. *Inclusive* object replication provides a method of ensuring that essential data will remain available, even after a node failure. By using the mechanisms that support atomic transactions, one can ensure the consistent behavior of actions, and can perform atomic, permanent, and serializable updates on data. The programming model technical report [Northcutt 88b] explains these Alpha robustness facilities more completely.

When failures occur in real-time systems, it is not sufficient to ensure that data and services remain available. It is also essential that the most vital tasks continue to execute within their time constraints. As described in the previous section, Alpha's time-driven resource management techniques are unique in their support of real-time systems where time constraints and available resources may change dynamically. In particular, the best-effort scheduling policies automatically adjust to overload conditions by ensuring that the most valuable threads continue to satisfy their timeliness requirements. This intelligent load shedding allows application and system performance to degrade gracefully in small increments. As with Alpha's other survivability mechanisms, the application designer controls (through time constraint specifications) how the system will respond to transient or permanent overloads.

Almost all of Alpha's mechanisms and services have been designed to facilitate the construction of survivable systems. The programming model, exception mechanism, and

resource management techniques are carefully integrated so that techniques such as object replication and atomic transactions fit cleanly into the system. The result is an operating system that makes it easier for application builders to design systems that operate correctly under stress. As an added benefit, the same Alpha facilities that allow a system to adapt to failures also allow it to evolve as requirements change. This evolvability ensures that Alpha-based systems will not only be able to survive, but will be able to adapt to changing environments and scenarios—potentially increasing the useful lifetime of the systems while reducing the cost of system enhancements.

2.4 Pre-Release Alpha Version 0.5

The Alpha application effort was begun while the operating system was still at an early stage of development. When GD began implementing the application, the Alpha group at CMU was working on pre-release 0.2 of the operating system. The final version of the ADSP runs under pre-release 0.5 (and later) of Alpha.

Pre-release Alpha 0.5 supports most of the kernel-level functionality (e.g., objects, threads, local and remote invocations, concurrency control, access control, and thread exceptions). In addition, an initial implementation of a best-effort algorithm is used for scheduling the application processor. However, the 0.5 system does not support many of the reliability mechanisms that would have simplified the implementation of the ADSP. In particular, the atomic transaction and inclusive replication facilities were not available.*

Pre-release 0.5 of Alpha also lacks many of the system-level services that are envisioned to be part of later implementations. Specifically, there are no user-level facilities for managing explicitly distributed services. Under most circumstances, distribution transparency is desirable; however, there are a few cases where the application prefers to manage distributed resources explicitly. It would be preferable to have the operating system provide services for managing these resources. Since Alpha 0.5 does not have a distributed systems services manager, an application-specific version was developed for the air defense system.

2.5 The Alpha Distributed Computer System Testbed

Alpha 0.5 runs on a special testbed that is designed to support distributed systems experiments. There are two copies of the Alpha prototype testbed—one at CMU, and one at GD. The sections below give a brief description of the testbed facility. A complete description of the design and construction of the testbed is given in *The Alpha Distributed Computer System Testbed* [Northcutt 88c].

The testbed facility at GD consists of two components—the *Distributed System Testbed* and the *Development and Control System* (See Figure 3). The Distributed System Testbed is the actual execution environment for the Alpha operating system and its applications. The Development and Control System is used for software development and for experimental monitoring and control.

* A specialized version of replication support was implemented for the ADSP; however, it was not available in time to be included as part of the public demonstration.

Development and Control System

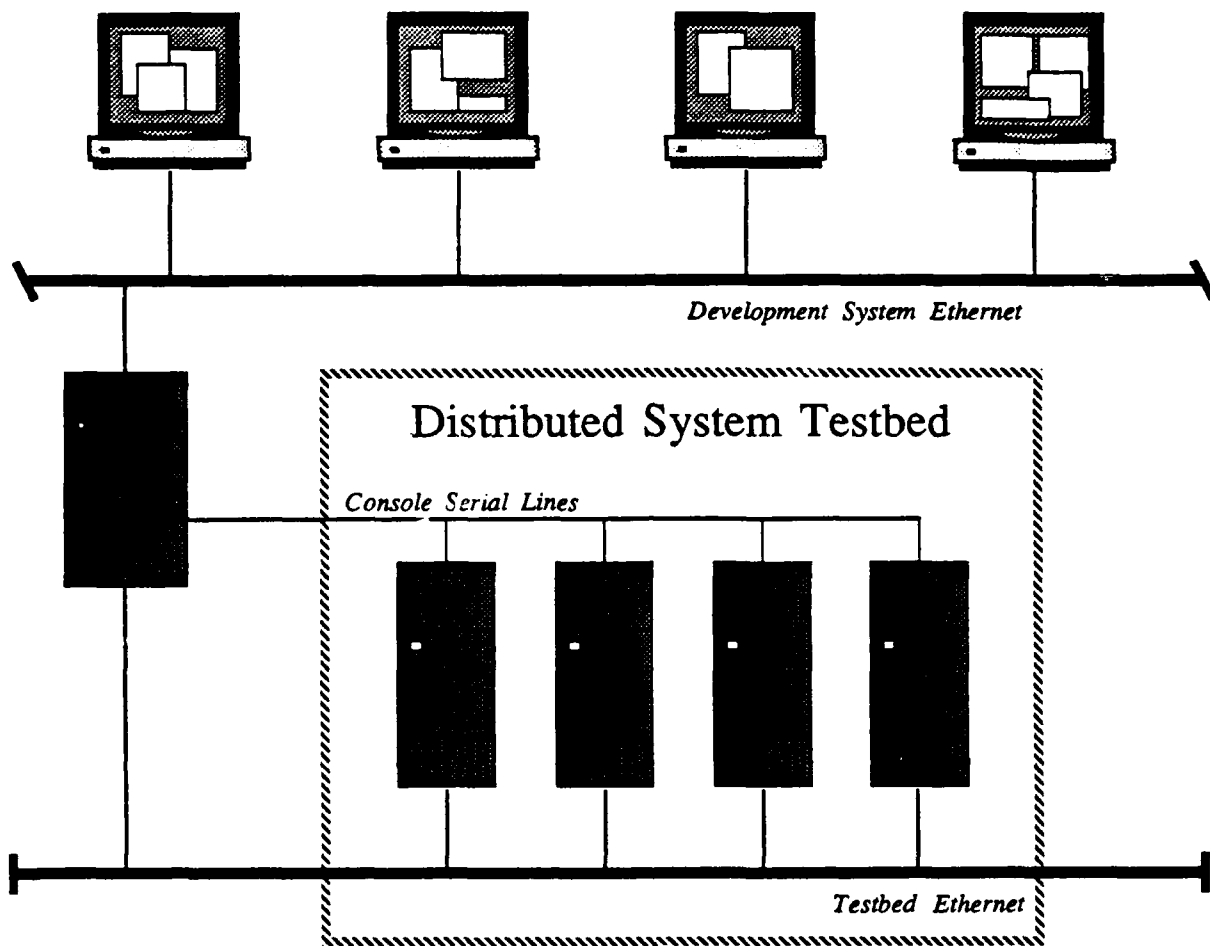


Figure 3: Alpha Distributed Computer System Testbed Structure

2.5.1 Distributed System Testbed

Alpha is designed to operate in a distributed environment containing one to 100 computing nodes connected by a local communications network. The distributed system testbed at GD contains four processing nodes connected by an Ethernet. An instance of the Alpha kernel executes on each of the nodes. The collection of nodes on which Alpha is running at any particular time constitutes the decentralized computer system on which applications execute.

The CMU implementation of Alpha has been designed to allow experimentation with hardware concurrency *within* the operating system. To support this type of experimentation, each node in the Alpha testbed contains multiple dedicated-function processors (see Figure 4). The Application Processor (AP) is a slightly-modified Sun Microsystems 2.0 CPU board that executes all application code as well as the Alpha kernel proper. The AP has 2 megabytes of local memory. The Scheduling Processor (SP) and Communications Processor (CP) are both (slightly-modified) Sun 1.5 CPU boards, each with

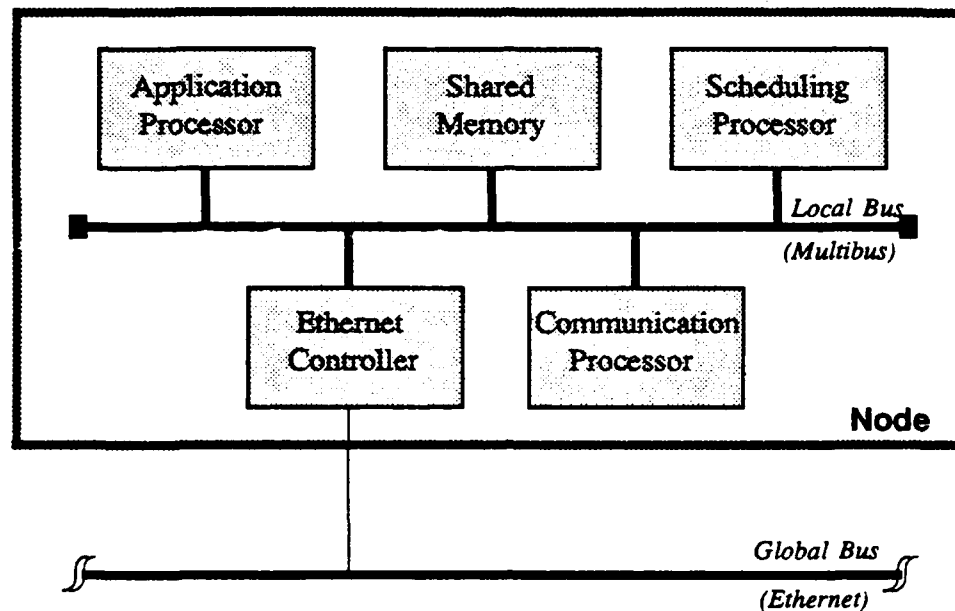


Figure 4: Alpha Testbed Node Architecture

256 kilobytes of memory. The processors communicate over a Multibus I backplane. In addition to the processors, each node contains a Sun Ethernet controller and 256 kilobytes of shared Multibus memory. A technical report [Northcutt 88d] on the Alpha kernel describes how each of these components is used by Alpha.

2.5.2 Development and Control System

The Development and Control System consists of several Sun-3 and Sun-4 workstations running Sun's version of the UNIX operating system. The workstations are connected by an Ethernet and use NFS to share files between machines. The Alpha group at CMU has implemented several tools that allow software developers to write and compile Alpha and Alpha-based applications in the UNIX environment. For a complete description of the Alpha software development tools, refer to the *Programming Utilities* technical report [Northcutt 88e].

The Development and Control System is also used for experimental monitoring and control. One of the development workstations serves as a gateway to the Distributed System Testbed. The gateway machine has two Ethernet controllers—one connected to the development system network and the other connected to the testbed network. This shared network link serves two purposes. First, it allows program images to be downloaded to the testbed from the development machines. Second, it permits run-time monitoring of and communication with the application while Alpha is running.

The gateway machine also has a serial line multiplexer connected to the console lines of each of the nodes in the Distributed System Testbed. This connection allows developers to control and monitor the testbed nodes remotely from the development workstations. Software developed jointly by CMU and GD provides window-based remote access to the testbed console lines.

3 Application Requirements

In order to learn as much as possible from the application effort, we needed to start with a sufficiently realistic and demanding set of system requirements. The first meetings between the Alpha group and General Dynamics personnel were dedicated to choosing and tailoring the scenario. The goal was to develop an application that was representative of real-time command and control systems, and that was suitable for use as an operating system technology demonstration.

The partnership with GD was vital in designing a realistic application. The team from GD was experienced in the design and implementation of air defense systems, and was familiar with the requirements of contemporary and future C²/BM applications. They had previously developed scenario simulations and user interfaces for similar applications. We relied mainly on their expertise to ensure that the chosen scenario was sufficiently realistic. When necessary, experts from other groups at GD were consulted to answer specific questions about system requirements.

The Alpha group wanted to highlight three key areas where the operating system provides unique support—decentralized computing, time-driven scheduling, and system survivability. Since proper handling of these areas often becomes more critical in larger systems, the “best” way to demonstrate the support would have been to construct a complete system including every conceivable activity. However, the complexity of large systems tends to obscure the relationship between a cause and its effects, making visual demonstrations difficult. Since the system had to be explained and demonstrated in a short, one-hour briefing, we needed easily-understood, highly-visible results. In addition, the scope of the effort was limited since only four months had been scheduled for the implementation of the entire system.

By the time the design meetings were completed (May 1987), we had chosen an air defense application and a set of requirements. We reduced the scope of the implementation effort by using simple mission planning algorithms and by partitioning functionality between the ADSP and the simulated environment.

The remainder of this chapter describes the scenario and the application requirements.

3.1 Scenario Description

The application scenario resembles a future coastal air defense environment. The primary purpose of the system is to coordinate the defense of the United States against airborne threats approaching over the Atlantic Ocean. Although inspired by the projected ADI environment, the details of the scenario are notional.

The application setting is the northeast United States air defense zone. Airborne, atmospheric threats approach the eastern seaboard at varying altitudes and velocities. The battle management system must use its assets to detect, track, and destroy the threats before they breach the integrity of the continental U.S.

The scenario assumes a wartime environment where approaching foreign craft are assumed hostile as soon as they are identified. The threats follow irregular flight paths and may change course at any time. The specific threats defined by the scenario are:

- manned bombers,
- manned cruise missile carriers,
- cruise missiles, and
- radar jammers.

Bombers and cruise missile carriers are large and relatively slow. They typically travel at altitudes where they can be detected at long range. In contrast, cruise missiles travel at high speeds and maneuver at low altitudes where they are difficult to detect with ground-based radar. The jamming devices constitute a secondary threat and have flight characteristics similar to the missiles.

The system identifies and tracks the hostile craft using sensor and reconnaissance facilities that include:

- over-the-horizon (OTH), ground-based radars,
- medium-range, coastal radar sites, and
- airborne early warning aircraft (e.g., AWACS).

The battle management system has three types of weapons available to intercept approaching threats. These weapons are:

- manned interceptors,
- medium-range guided missiles, and
- point-defense, surface-to-air missiles (SAMs).

The manned interceptors are fighter aircraft armed with heat-seeking air-to-air missiles. Guided missiles are faster than manned interceptors and have automated terminal guidance. However, like manned fighters, they require external guidance until they are within range of their target. SAM's are available as a last-resort coastal defense. They have very limited range and must be aimed and activated by the ADSP.

The scenario does not define absolute performance characteristics for the threats, sensors, or weapon systems. In most cases, experiments were carried out using capabilities that were intended to represent a late-1990's environment. However, no attempt was made to guarantee the realism of the numbers used for altitudes, speeds, radar ranges, and so forth. Instead, emphasis was placed on the aspects of the scenario that define the types of activities the system must perform and the requirements it must fulfill. The following sections concentrate on these requirements.

3.2 Application Processing Requirements

The primary requirement of the air defense system is to keep hostile targets from entering the airspace of the continental U.S. The activities required to achieve this goal can be classified into three main areas: *tracking*, *mission planning*, and *weapon systems control*. At this high level, these functions are similar to those found in many C²/BM applications. The following sections detail the application processing requirements in each of these areas. Figure 5 outlines their basic functionality.

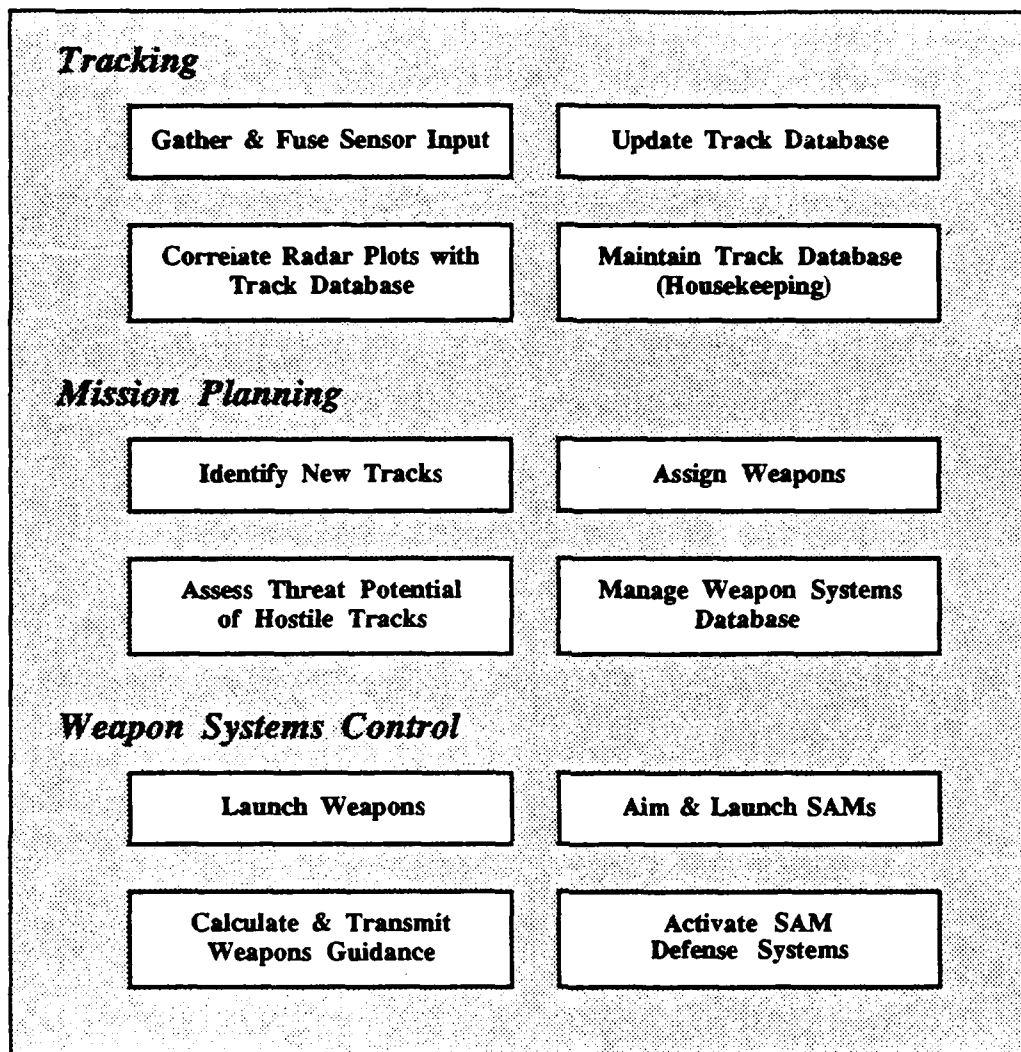


Figure 5: High-Level System Functions

3.2.1 Tracking

The first requirement of the air defense system is that it be able to provide information about the aircraft and other airborne platforms in the area of interest. The system needs to maintain a tracking database that contains position, velocity, and identity information for each target in the system. The database should also contain information about position histories to aid in track visualization and plot correlation.

To maintain the tracking database, the system must collect data from various sensor systems. This information must be fused into radar reports that contain the position, altitude, and identity (if known) of each object detected. These fused reports should then be transmitted to the other tracking functions for incorporation into the database.

When a sensor report is received, it must be correlated with previous information to determine whether or not the report is associated with a known flight path (or track). If so, the associated track information must be updated. If the report does not correlate with an

existing track, the data should be entered in the database as an uncorrelated report. If subsequent radar plots match with the uncorrelated report, a new track entry should be created and the mission planning functions should be alerted of its existence.

There is also a certain amount of tracking database maintenance that must be performed to ensure that stale information does not cause errors in tracking or target identification. The system must periodically scan the tracking information, purging old tracks and uncorrelated reports that have not been updated within a certain time.

3.2.2 Mission Planning

When the tracking functions discover a new track, the mission planning components identify the track and determine whether or not it is a threat. Track identification may use such information as track origin (did it originate from a commercial U.S. airport?), velocity (is it too fast to be a bomber?), and identify-friend-or-foe (IFF) transponder information.

If the track represents a threat, its threat potential must be evaluated. This assessment is based on such considerations as threat type (*e.g.*, bomber or cruise missile), position (*e.g.*, distance from the coast), speed, and direction. Threats become more serious as they approach the coastline and interior of the U.S.

The system must determine the best choice of weapon system and locate the base nearest the target where the system is available. Defensive weapon assets include manned interceptors (*e.g.*, F-15, F-16), long-range guided missiles, and point-defense surface-to-air missiles. The weapons pairing decision is based on such considerations as the estimated intercept time for each type of weapon. Once chosen, the weapon must be allocated and passed to the weapon systems control functions for launching and in-flight control.

The mission planning component must also maintain situation assessment and resource availability information. In particular, the system must maintain a database containing, for each base, its location and its current inventory of weapons. It is also important to maintain a listing of all active target/weapon pairings.

In a real battle management system, the mission planning function would likely be carried out by a group of people using information provided by sensor data and intelligence sources. Since the ADSP is designed to be able to run without human intervention, each of the mission planning functions is carried out by the computer using simple heuristics. In a more advanced demonstration, these heuristics could be replaced by expert systems designed to operate in concert with a human advisor.

3.2.3 Weapon Systems Control

When a weapon and launch site have been chosen, the actual launch command must be issued along with initial guidance information. Once the interceptor is in the air, its progress must be monitored, and its course frequently updated to compensate for changes in the direction of the target and for navigational inaccuracies. If for some reason a weapon does not reach its designated target (*e.g.*, it is destroyed en route), the mission planning system must be notified so that replacement weapons can be allocated. In addition, for manned interceptors, the interceptors must be guided back to an airbase after their encounter with the target.

The weapons control functions are also responsible for the operation of the terminal defense system (coastal SAMs). The coastline must be periodically scanned for hostile craft that have penetrated the outer defense systems. When threatening tracks are discovered, a SAM site must be chosen and activated to intercept the target.

The application requirements make several simplifying assumptions with regard to weapon systems. In particular, weapons are presumed to have perfect kill ratios. That is, if a weapon is successfully launched and guided to within a set distance of its target, it is assumed to destroy the target. Although it would be trivial to model a kill ratio less than one, the behavior of the system would be more difficult to understand in a visual demonstration (since one could not tell if an intercept failed because of an Alpha policy decision or a random miss probability.)

3.3 Application Timeliness Requirements

The different activities that compose the application have a wide variety of timeliness requirements. The majority of the application tasks are aperiodic and have soft time constraints that vary with time and external inputs. Other activities are roughly periodic with soft time constraints that are fixed at compile time. Still others are aperiodic with hard deadlines. In addition to timeliness concerns, each of these tasks have different relative importances that may change dynamically. Three typical classes of timeliness requirements are represented by the plot correlation activity, the weapons guidance activities, and the SAM launch sequence. These requirements are outlined in the remainder of this section. The time-value functions corresponding to these requirements are discussed in Section 8.2.

3.3.1 Radar Plot Correlation

Radar plot reports arrive from the sensor system in *frames*. The data for each frame should be correlated with the tracking database before the next frame arrives. Since frames arrive at roughly periodic (six second) intervals, the plot correlation activity is approximately periodic with a six second deadline. However, the task is not truly periodic in the classical sense. In particular, the task is initiated by an external event (the arrival of a new frame) instead of being activated periodically. Also, plot correlation does not have a hard deadline. If the processing of a frame exceeds the six second time, it is still important to complete the processing of that frame. Only if the processing is not completed in two frame times should the attempt be aborted.

Because of the fundamental importance of tracking to the application, the plot correlation activity has one of the highest task importances in the system. The track database maintenance activity is also roughly periodic with a six second frame time. In contrast, however, the maintenance activity has a very low relative value. This example clearly demonstrates how a task's importance can be independent of its critical time.

3.3.2 Weapons Guidance

Weapons guidance is one of the most time-consuming activities in the system. Each guidance calculation requires a significant amount of processing time, and several interceptors may require guidance during a short interval. For the air defense system, weapon headings should be updated when approximately one fourth of the predicted time-to-intercept has elapsed. Since the time-to-intercept depends on the distance between the

interceptor and its assigned threat, the guidance activity is fundamentally aperiodic and dynamic. It is impossible to know the critical times when the application is compiled.

The critical time for guidance updates indicates the time when course corrections are preferred. If excess resources are available, additional guidance updates may be issued (potentially decreasing the intercept time). Likewise, if the system is overloaded, guidance for interceptors that are still far from their targets can be delayed without a significant penalty. This flexibility is expressed by a variable "hardness" of the guidance time-ness requirements. When the time-to-intercept is large, the time constraint can be fairly soft since using slightly outdated tracking information will not significantly affect the intercept. However, as the intercept time nears, it becomes increasingly important to complete the course update prior to the critical time.

The importance of the guidance activities also varies. The value of completing an intercept is tied directly to the threat potential of the target. In the prototype system, the threat potential is based primarily on distance to the coastline and the type of threat. As with the plot correlation activity, the importance of a guidance task is independent of its critical time.

3.3.3 SAM Control

The weapon control program for the surface-to-air missiles is one of the few activities in the system with a strict hard deadline. The launch sequence models a system using an aim-aim-fire algorithm—where two successive sightings are used to aim the missile before firing. The aim-aim-fire sequence must be completed by its deadline or the launch is aborted and a new launch sequence initiated. In addition, a successful launch must be made before the threat leaves the range of the SAM battery.

Since the coastal SAMs are the final line of defense in the system, a failure in the launch sequence will mean that the battle management system has failed to meet its primary objective. Consequently, the SAM control tasks have the highest value to the system when they are active.

It is interesting to note that although the SAM control tasks are among the few activities with hard deadlines, they are strictly aperiodic. The SAM batteries are only activated when a hostile craft approaches within a certain distance of the coastline.

3.4 Application Survivability Requirements

One of the fundamental constraints of the air defense application is that the system must continue to fulfill its mission requirements even if one or more of the processing nodes fails. For the air defense application, fulfilling the mission means that coastal integrity is preserved even after nodes fail. In addition, the system must be capable of re-integrating nodes that have been repaired after a failure.

In order to respond when failures occur, the system must ensure that essential data remains available after a node failure. In particular, essential repositories such as the tracking database and the resource (weapons) database must survive failures so that their data will continue to be available. Not all data must be saved however. For example, it is acceptable for a single frame of radar reports to be lost when a node fails. Since radar reports will continue to arrive, the lost reports will quickly be replaced by fresh ones.

In addition to mission data, essential services must also continue to be available after failures. All vital tracking, mission planning, and weapon systems control functions must be obtainable after a node has failed. In many cases, it is not necessary to replicate the functionality so long as new service providers can be created dynamically to replace those that are lost.

Finally, it is important that the system not "forget" any important activities that were in progress when a node failed. In particular, weapons that were being guided by a lost node must have new guidance activities created on the nodes that remain available. Other system activities, such as database maintenance, also need to be restarted if they are prematurely terminated.

Ultimately, the real survivability requirements are that, as much as possible, the system continue to satisfy the processing and timeliness requirements outlined in the previous sections. In some cases, the combination of limited resources and additional exception processing after a failure might lead to system overloads. In such circumstances, the system should reconfigure itself or shed less vital activities until the situation is brought under control. Because the defense application operates in a highly dynamic environment, it would be impossible to construct a rigid system that could adapt to all of the failure scenarios. By designing a system that can respond flexibly, it is possible to continue satisfying mission requirements under a wide range of load and failure conditions.

4 Experimental Environment

The high-level architecture of the experimental system is shown in Figure 6. The system is divided into two major sections—the air defense system prototype (ADSP) and the experimental support environment. The ADSP runs under Alpha on the distributed computer system testbed. It implements most of the functions of a real air defense system including the tracking, mission planning, and weapon systems control functions outlined in the previous chapter. The remainder of the system supplies a simulated execution environment for the air defense system and provides monitoring and control functions for a human experimenter. These support functions run on Sun workstations under UNIX. The support environment is divided into several components—the Scenario Simulator, the Experimental Control Console, the Operator Console, and the Communication Interface.

As much as possible, the design team tried to preserve the division between the prototype system and its support environment. The goal was not to *simulate* an air defense system, but to build a (limited-function) air defense system prototype that operated in a simulated environment. Chapters 5, 6, 7, and 8 describe the design and operation of the ADSP. The remainder of this chapter describes the environment in which it operates.

4.1 Scenario Simulator

The Scenario Simulator is responsible for providing the environment in which the ADSP operates. It is based on a validated simulator developed by GD for an earlier project. The program simulates the movement of all hostile and friendly tracks and implements the radar coverage model that determines which tracks are visible to the defense system. It is also responsible for detecting “kills” that occur when an interceptor reaches its target. Every six seconds, the simulator computes new positions and velocities for the aircraft based on flight paths and guidance information. For those tracks that are visible, it transmits radar reports to the ADSP. Each of the reports contains the position, velocity, and any available identification information for the target. The actual position calculations performed by the simulator vary depending on the type of track.

Hostile flight paths are loaded into the simulator during system start-up. Flight paths are stored as a list of course changes. Each course change contains an activation time, an initial position, and a velocity vector. For each update, if the current simulation time equals the activation time for the next course change, the new position and velocity are set to the stored value. By storing both position and velocity for the course change, the simulator avoids cumulative round-off effects and ensures that hostile craft will follow their intended path. If a course change is not scheduled, the new position is computed based on the previous position and velocity.

For friendly interceptors, guided missiles, and radar aircraft, launch and guidance commands are received from the ADSP while the simulator is running. During each simulator update, new positions are computed for existing friendly tracks. The new positions are based on the previous position and velocity. After computing the new positions, any guidance updates are applied to adjust the speed and heading of the track. Finally, launch commands are processed by adding new interceptors, missiles, or AWACS to the simulation with the initial positions and velocities given in the launch commands.

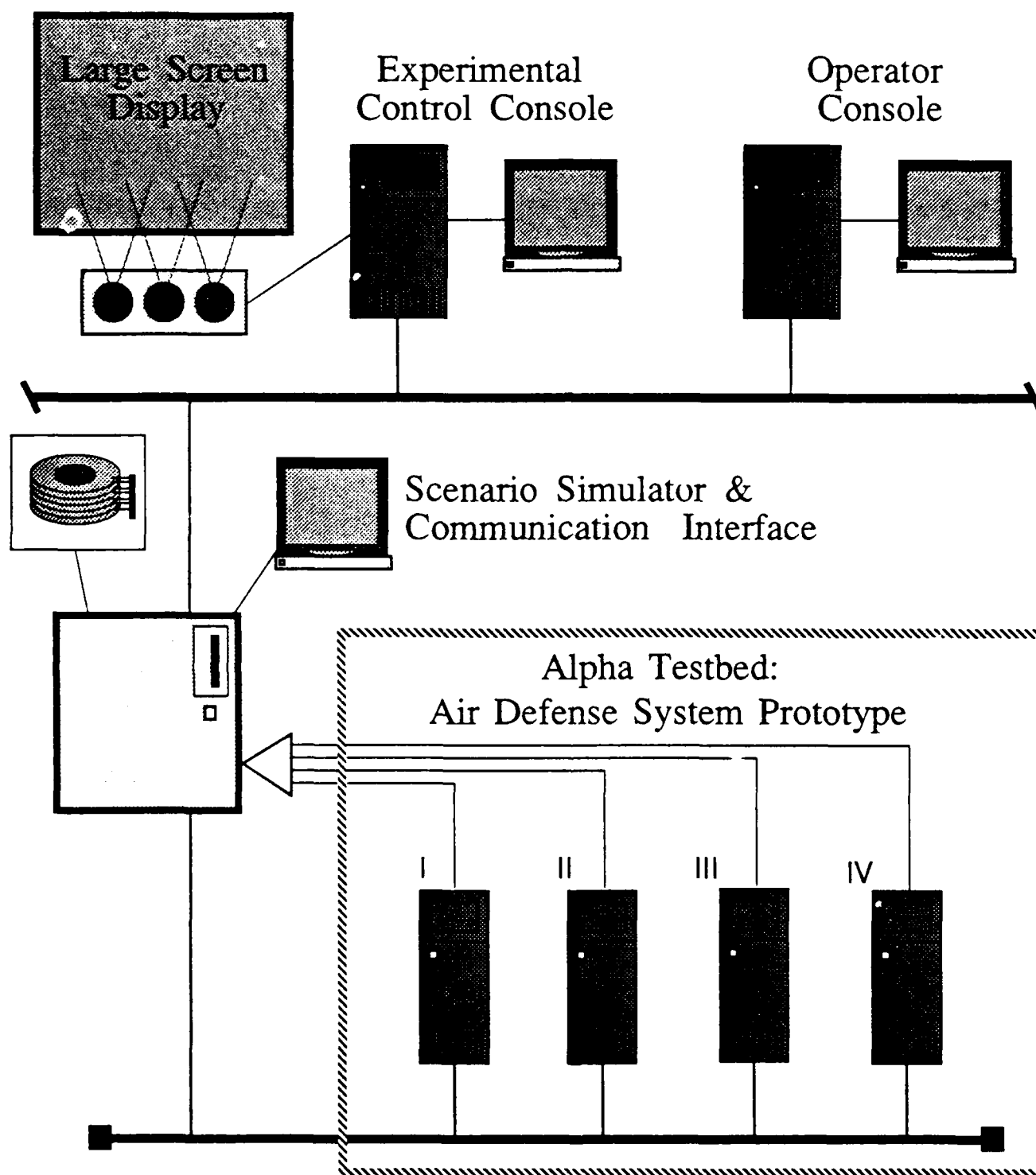


Figure 6: Overall System Architecture

After computing new positions and velocities for each track, the simulator applies a radar coverage model to determine which tracks are visible to the defense system. Each track position is checked to determine if it is visible to any of the radar sites (either ground-based or airborne). Each radar site has a cone-shaped coverage where the range of the radar increases with altitude. The simulator is implemented so that only one radar report is issued for any track in the system. This restriction means that no fusion processing is required in the ADSP. The design team chose not to implement sensor fusion for a variety of reasons. Chief among these reasons was the limited implementation time available.

In addition to course calculations, the simulator is responsible for detecting successful intercepts or "kills." When an interceptor or missile approaches within its kill range of a hostile craft, the enemy target is removed from the simulation. If a missile makes the kill it is also removed. The simulator does *not* issue any form of message when a kill happens. It is the responsibility of the ADSP to assess that a kill has occurred and initiate activities such as directing manned interceptors back to their airbases.

4.2 Operator Console

The Operator Console is analogous to the actual control console that a C²/BM system operator would use. It provides an air situation display including a map, track information, resource locations, radar coverage, and other information useful in mission planning and control. Button panels allow the user to adjust the type and detail of information displayed. The console allows the operator to request detailed information about particular tracks or target/weapon pairings. In a real system, the operator's console would be highly interactive—requiring the user to input a variety of weapons assignment and control commands. For this application effort, the battle management system operates autonomously. The only commands (other than display control commands) issued by the human operator are those to launch airborne early warning aircraft.

The Operator Console receives display lists from the ADSP and converts the lists into an air situation display. This type of connection is similar to those used in many existing command and control systems. Ideally, however, we would have preferred to drive the Operator Console directly from one of the Alpha processing nodes. Unfortunately, the distributed system testbed does not support high-resolution color graphics, so it was necessary to implement the console on a Sun-3 workstation.

The display itself is based on the latest user interface technology and uses multiple display windows and virtual control panels that are activated by a "mouse" pointing device. The display is implemented under SunView^(tm) using SunCORE^(tm) graphics primitives. A sample screen snapshot is shown in Figure 7. The actual display uses color to enhance the contrast between different types of tracks and geographical features.

The Operator Console provides a variety of commands that control the type of information shown on the air situation display. Individual types of tracks can be turned "ON" or "OFF" to reduce display clutter. The following track types can be individually selected:

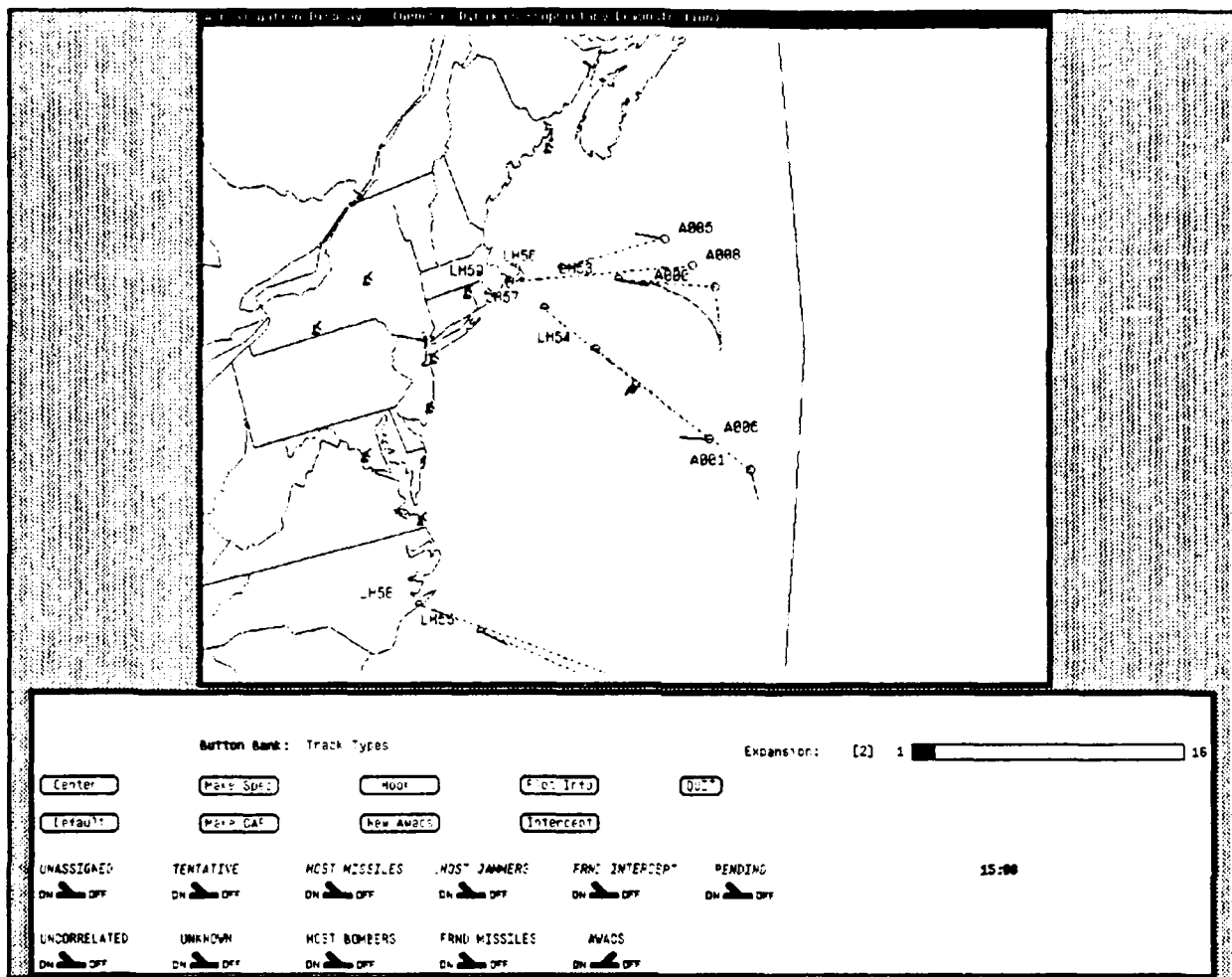


Figure 7: Operator Console Display

<i>Uncorrelated Reports</i>	<i>Hostile Missiles</i>	<i>Friendly Missiles</i>
<i>Tentative Tracks</i>	<i>Hostile Bombers</i>	<i>Friendly Interceptors</i>
<i>Unknown Tracks</i>	<i>Hostile Jammers</i>	<i>Friendly AWACS</i>

The console also allows the operator to control how much information is displayed about the different tracks. In particular, the user can choose whether or not to display the track ID, track block (position and velocity information), position histories, and pairing lines (lines that indicate which hostile target a friendly interceptor or missile is pursuing). The operator can also obtain additional information about particular tracks by selecting them with the mouse. The system will display detailed identification and tracking information as well as pairing information and predicted intercept paths for the selected track(s).

In addition to track information, the air situation display can present a variety of tactical data. Options allow the user to display military bases, major cities, SAM sites, and radar sites. For the radar sites, the system will display radar coverage rings for altitudes of 1000, 10000, and 25000 feet. The operator may place and selectively display reference

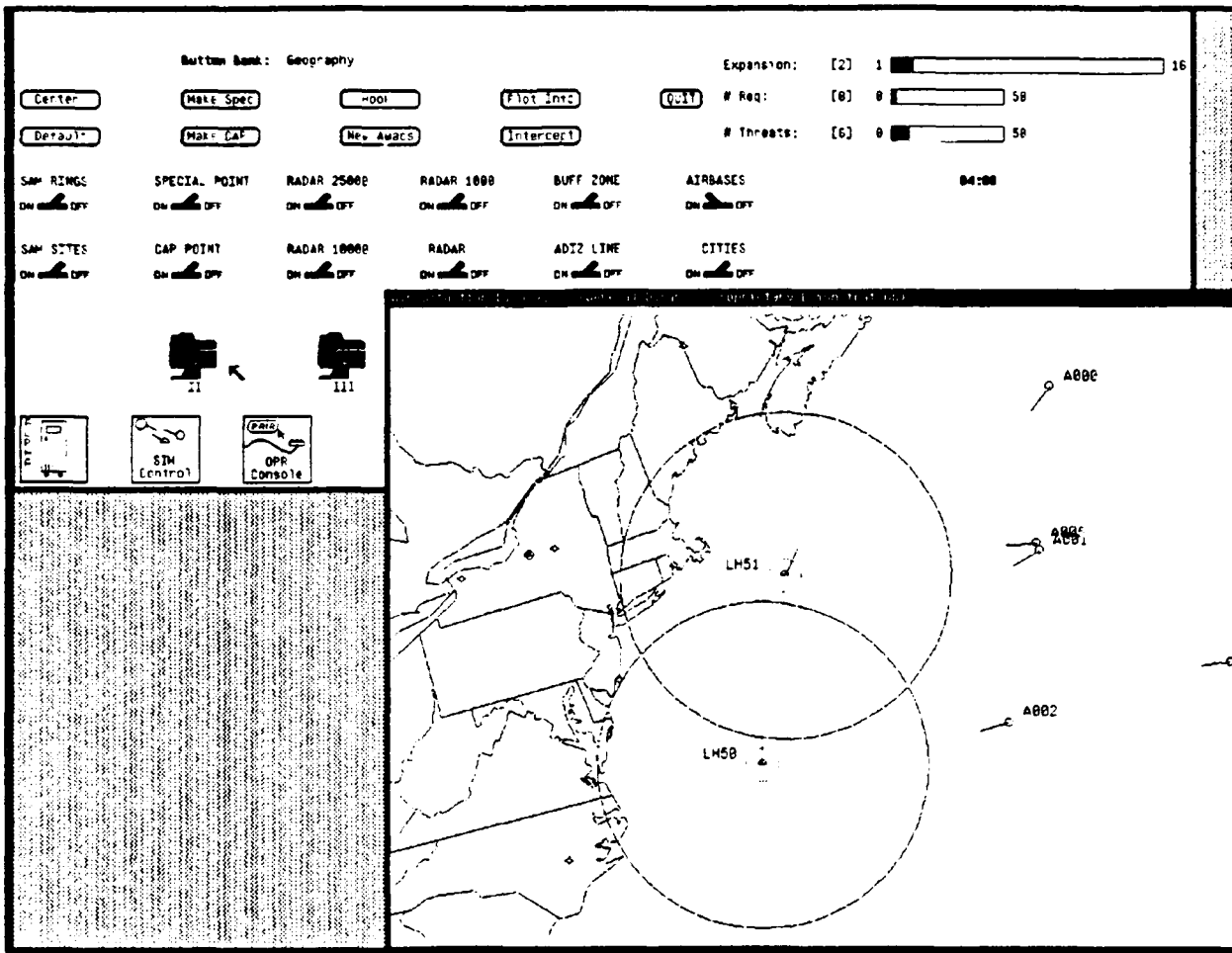


Figure 8: Experimental Control Console Display

points to aid visualization. Finally, the entire situation display can be expanded around arbitrary locations and displayed at magnification factors from 1X to 16X.

The only battle management function performed by the human operator is the launch of airborne early warning aircraft (AWACS). The operator uses the mouse to select the desired coverage area and issues a request to the battle management system. The ADSP then sends a launch command to the Scenario Simulator and guides the AWACS to the desired patrol area. Once in the desired region, the air defense system automatically guides the AWACS in an oval pattern. Since new AWACS planes are automatically added to the simulator model, the operator is able to control the scope of radar coverage available to the system.

4.3 Experimental Control Console

The main purpose of the Experimental Control Console is to allow the experimenter to monitor the behavior of the application and to control the simulated environment. For public demonstrations, it is connected to a large-screen display device so that a large audience can follow the scenario. The Experimental Control Console is similar in appearance to the Operator Console. Like the Operator Console, it provides an air situation dis-

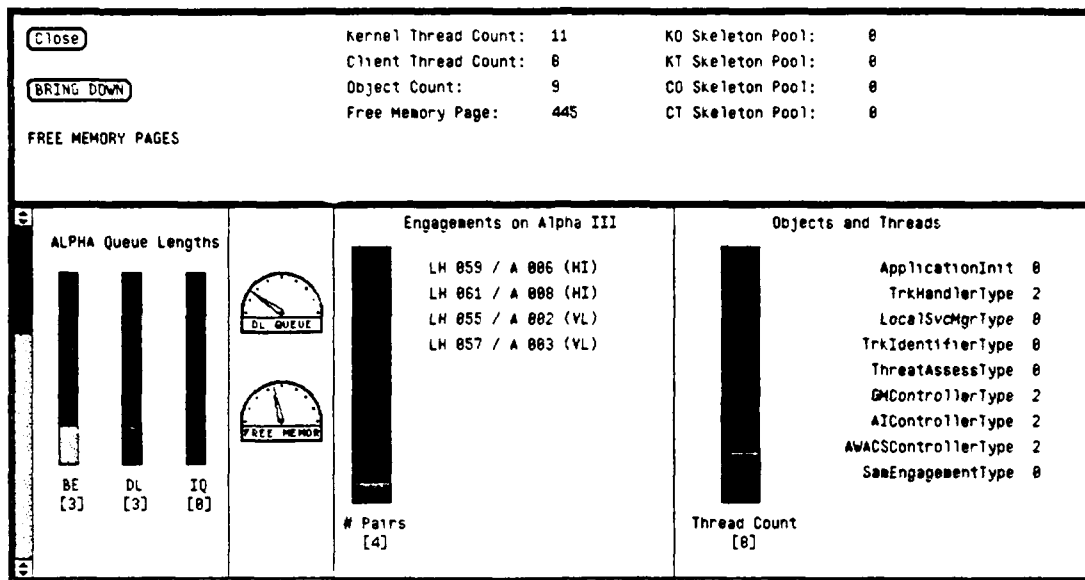


Figure 9: Node Information Display

play and resource information. However, the Experimental Control Console displays *reality* as defined by the Scenario Simulator, not as perceived by the defense system. Therefore, the tracks displayed are not subject to radar coverage limitations or time delays due to track processing.

The air situation display of the Experimental Control Console is almost identical to the Operator Console (see Figure 8). Unlike the Operator Console, the Experimental Control Console has an additional row of icons at the bottom of its button panel that monitor the health of the other components in the support system (e.g., Scenario Simulator, Operator Console, Communication Interface). There is also a row of icons indicating which of the Alpha nodes are currently running. When a node fails (is turned OFF during the demonstration) it is indicated on the Experimental Control Console.

The Experimental Control Console also allows the user to monitor the activity and performance of each node in the distributed system testbed. By selecting an Alpha node icon with the mouse, the experimenter activates a node information display. Node information displays (see Figure 9) provide resource information such as scheduler queue lengths and amount of available memory. They also list the weapon pairings being controlled by the node, the types of Alpha objects resident on the node, and the number of threads active in each object.

The Experimental Control Console is also used to control the Scenario Simulator. During system start-up it is used to script flight paths for hostile aircraft and missiles. A special display window allows the experimenter to enter flight paths graphically with the mouse. The scenario designer can specify hostile missile launch points and hostile warhead detonation points. Once the system is running, the console can be used to raise or lower the number of threats using control sliders. The sliders, located at the top-right of the button panel, display the current number of threats and allow the user to specify a desired maximum.

4.4 Communication Interface

The Communication Interface controls the flow of information between the ADSP and the different components of the support environment. Information such as radar reports from the simulator, display list information for the Operator Console, interceptor guidance information from the weapon controller, and system monitoring information from the testbed is transmitted over this interface. The interface has two components—the message multiplexer and the Alpha/UNIX communications link. The message multiplexer distributes and routes messages between the Scenario Simulator, Operator Console, and Experimental Control Console. The communications link provides the actual data transfer path between the Alpha nodes and UNIX.

The message multiplexer provides a generalized data distribution service. The different components of the support environment communicate using tagged messages. Each message consists of a tag indicating its type (e.g., radar report, guidance command) followed by the body of the message. The contents and length of a message are defined by a "C" language data structure for each message type. The multiplexer is connected to each of the components in the support environment via UNIX sockets. Messages arriving at the multiplexer are routed to one or more destinations based on a routing table that is loaded during system initialization. The message multiplexer also performs optional processing on each message when format conversions are needed between programs. The flexible routing scheme was especially valuable during the development and debugging stages since it allowed test stubs to be installed in place of components that were still under development.

The Alpha/UNIX communications link provides a reliable bidirectional connection between an Alpha object and a UNIX socket. On the Alpha side, a kernel object encapsulates the Alpha device drivers and provides operations to send and receive messages. On the UNIX side, a process handles low-level details and shuttles data to and from a socket that is connected to the message multiplexer.

The communications link operates either over an RS422 serial line or over the testbed Ethernet. Because Alpha supports an object interface, it is easy to change between the two links. The SerialIO and NetworkIO objects (that encapsulate the links) provide the same external interface, so it is only necessary to change the object reference that is distributed to other objects in the system. In the final version of the air defense system, the change between the RS422 and Ethernet links can be made by changing *one* line in the application code.

5 Air Defense System Design

The air defense system was the first significant application designed for and implemented using the Alpha programming model. Although there was no established methodology for developing Alpha-based applications, the design team found conventional software engineering principles to be well-suited for use with the object/thread paradigm. Many of the fundamental concepts of software engineering (*e.g.*, abstraction, information hiding, and modularity [Fairley 85]) are directly supported by the object model. In fact, GD found that Alpha's programming model simplifies the software design process by allowing designers to move directly from a functional decomposition of the problem to an object-based implementation. Since no conversion step was required to translate the natural system structure into "foreign" notions such as client/server processes or rate groups, the requirements specification led directly to an implementation that was easy to understand and simple to modify as requirements changed.

Starting from the system requirements outlined in Chapter 3, team members from GD and CMU used a top-down approach to develop a design for the air defense system. This design was subsequently enhanced and implemented by the group working at GD.

5.1 Design Approach

The application effort provided the participants with significant experience in the design and development of distributed, real-time systems. By applying some fundamental software engineering principles to the design of the ADSP, the designers developed an informal approach to creating applications for the Alpha programming model. Although the technique is not sufficiently refined to classify it as a methodology, it does offer guidelines and insights that can be used in developing future systems.

Using the system requirements as a basis, team members from GD and CMU used a top-down approach to design the air defense system. The design process can be broken down into the following components:

- Identify major system functions and activities
- Specify objects that provide the necessary functions
- Define threads that carry out the required activities
- Assign time constraints to threads
- Distribute the application across nodes

The first step in designing the air defense application was examining the requirements to identify the major functions and activities. The designers found it useful to represent the major components as (potentially ill-defined) objects and threads. Attempting to specify functionality and operations for these "objects" often revealed natural boundaries where components should be further specified or divided.

Once the major system functions had been defined, standard software engineering principles were used to modularize the system into well-defined objects and operations. The specification of the objects was a major part of the design effort. It was also the part most amenable to conventional design methods. Designers used techniques such as data abstraction and functional decomposition to identify object boundaries and specify appropriate operations. Well-defined metrics such as *coupling* and *cohesion* were used to

aid with the division and to identify parts of the system where more design work was needed.

While the system functions were being decomposed into object specifications, work continued on defining the application threads. Threads correspond to activities performed by the system. Describing activities as threads provides a way of specifying how compound functions are accomplished using many objects. Threads travel transparently between objects and nodes and may execute concurrently. The designers considered several alternatives when deciding which activities should have separate threads. The general guideline adopted was that threads should be defined for activities that were logically independent and/or that had independent timeliness requirements. Chapter 7 describes the application threads that were chosen.

Once the threads had been defined, rules were developed for determining their time constraints. In some instances, the timeliness requirements were static and easy to define. In others, the time constraints were both dynamic and data-dependent. In the more complex cases, the general shape of the time-value function was chosen based on the timeliness characteristics of the task. Specific rules were then developed for calculating the parameters that define specific time-value functions (such as critical time and maximum value) from run-time information. Section 8.2 describes the time-value functions that were eventually derived.

To a large extent, the object type specifications were developed without considering how the system would be distributed across multiple nodes. Since Alpha provides transparent distribution, the objects could be defined as if the application were going to execute on a centralized system. Distribution issues became important when considering the survivability, availability, and concurrency requirements of the application. Survivability requirements determined which objects would be replicated, while availability considerations influenced the number and placement of object instances. The desire to improve performance through application concurrency influenced how threads were distributed among the nodes. Section 8.1 explains how these requirements were used to develop a plan for distributing the objects and threads that comprise the air defense system.

5.2 Functional Decomposition

The application requirements outlined in Chapter 3 divide the ADSP into three major components—tracking, mission planning, and weapon systems control. The first step in designing the air defense system was analyzing these areas to identify the major functions and activities. The purpose of identifying these components was not to develop an exhaustive or precise list. Rather, the goal was to provide a basis from which further design work could proceed. The following paragraphs briefly outline the results of the analysis.

Tracking revolves around the maintenance of a database containing positions, velocities, and other information about airborne objects. The primary tracking activity involves:

- collecting radar reports from the Communication Interface,
- correlating the radar reports with the tracking database,
- updating the tracking database to reflect the new information, and
- transmitting display information to the Operator Console.

The tracking component is also responsible for identifying new tracks that enter the coverage zone and for purging old ones that have either left the zone or been destroyed. Additional tracking functions allow other parts of the system to retrieve and update information about particular tracks.

The mission planning component determines and implements the appropriate response to new tracks. The major functions include:

- determining the identity of new tracks,
- assessing the threat potential of hostile tracks,
- assigning appropriate weapons to intercept hostile craft, and
- initiating weapons launch and guidance sequences.

Every new track that enters the system must be identified. The remaining mission planning tasks are performed only for hostile tracks. Identification and threat assessment rely on information from the tracking database. To support the weapon assignment function, the mission planning system maintains a separate database containing information on available weapon systems and active target/weapon pairings.

Weapon systems control involves:

- launching weapons and
- guiding weapons to their designated targets.

Specific versions of the control functions are required for each type of weapon. The long-range weapons (*i.e.*, airborne interceptors and guided missiles) require frequent course corrections to ensure that they successfully intercept their targets. AWACS aircraft also require course updates to maintain their oval flight pattern. The short-range SAMs only require initial guidance. However, a separate weapon control activity is required for detecting when targets approach within range of the SAM batteries.

The tracking, mission planning, and weapon systems control functions exist primarily to satisfy the application processing requirements. Additional functions are needed to fulfill the system survivability requirements. Specific functions dedicated to distribution and survivability are responsible for tasks such as:

- distributing the processing load among multiple nodes, and
- helping the system recover from node failures.

Finally, because the ADSP is an experimental prototype, other functions are required to integrate the system with its operating environment. These support functions include:

- handling communications with the experimental environment, and
- supplying system monitoring information to the Experimental Control Console.

Although this list of activities and functions identified by the requirements analysis was not exhaustive, it provided the basis for defining the objects and threads that would compose the air defense system prototype. The following two chapters describe the actual objects and threads that were chosen.

6 Air Defense System Objects

The functional decomposition given in Section 5.2 provided the basis for choosing the object types. After the major system functions had been identified, related functions were grouped into objects using standard software engineering techniques. Figure 10

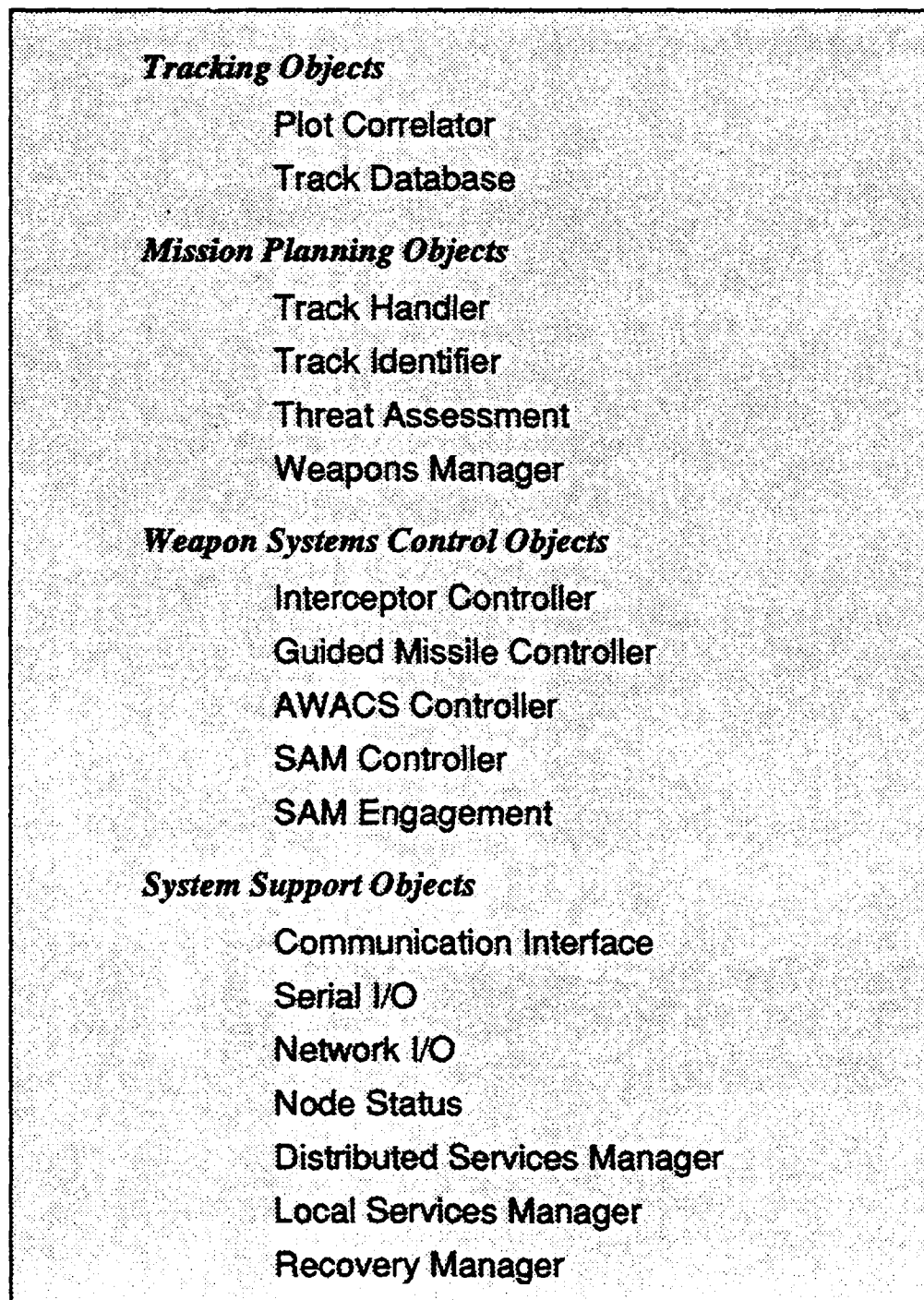


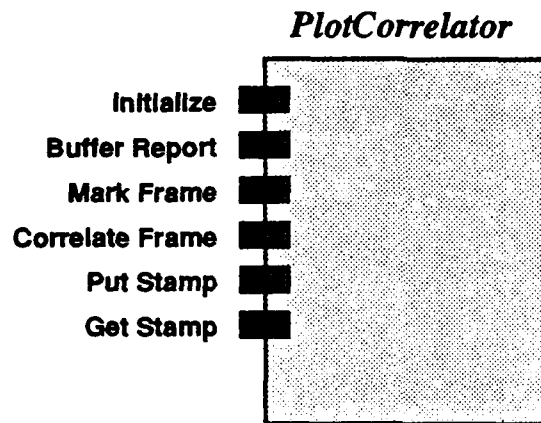
Figure 10: Air Defense System Object Types

lists the object types that were chosen. The remainder of this chapter describes each of the objects in greater detail.

In the remainder of the text, object and operation names are set in italic type (e.g., *PlotCorrelator*). At times, spaces have been inserted in object or operation names to improve legibility. Operations within a specific object are referenced using a "dot" notation (e.g., *Object.Operation*, *PlotCorrelator.Initialize*).

6.1 Tracking Objects

6.1.1 Plot Correlator

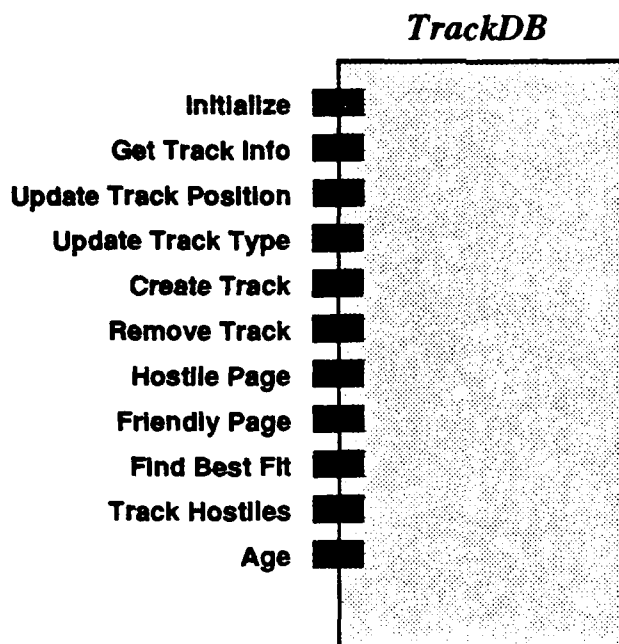


The *PlotCorrelator* object correlates incoming radar reports with existing tracks in the tracking database (*TrackDB*). Incoming radar reports are buffered in the *PlotCorrelator* until an end-of-frame marker is received. The entire frame of data is then correlated with the database contents. Reports that do not correspond to existing tracks are entered into the database as uncorrelated reports. When three consecutive reports are received for a new track, its type is changed to "unknown" and a new thread is created to handle its identification and processing.

The operations defined on the *PlotCorrelator* are:

- *Initialize* — initializes internal data structures and creates a thread in the *CorrelateFrame* operation.
- *Buffer Report* — buffers a radar report in the local queue.
- *Mark Frame* — marks the occurrence of the end of a radar frame.
- *Correlate Frame* — correlates an entire frame of radar reports. The operation blocks until a full frame has been received. It then correlates the reports (see *TrackDB.FindBestFit*), creating new tracks for plots that do not match, and creating new threads to handle unknown tracks as they enter the system. When processing for the frame has finished, the operation loops back and waits until the next frame of data has been received.
- *Put Stamp* — stores a timestamp in the timestamp buffer.
- *Get Stamp* — retrieves the next timestamp from the timestamp buffer.

6.1.2 Track Database



The track database (*TrackDB*) object encapsulates the tracking database and all of the routines that manipulate the database. It maintains position, velocity, and identity information for all known tracks. Tracks can be created, removed, and updated (with new position or type information). The database also defines special operations used to correlate radar reports with existing database entries and to scan the coastline for hostile aircraft.

The operations defined on the *TrackDB* are:

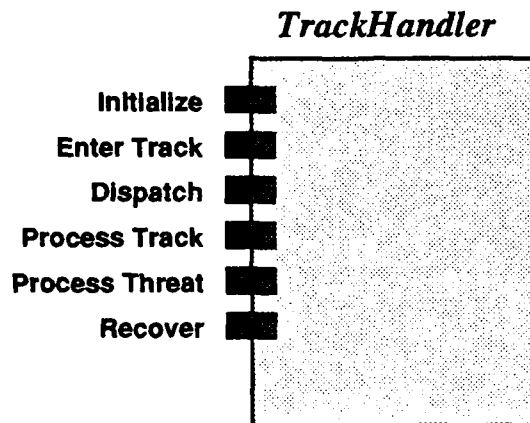
- *Initialize* — initializes the database and starts a “housekeeping” thread in the *Age* operation.
- *Get Track Info* — retrieves information about a specific track (based on its *track_id*).
- *Update Track Position* — sets a track’s position and velocity to the given values. This operation is used by the plot correlation function to update the database when new radar data is received.
- *Update Track Type* — sets the “track type” entry for the specified track. This operation is used by the new track processing functions to indicate whether a track is a hostile bomber, friendly interceptor, friendly missile, etc.
- *Create Track* — creates a new track entry in the database.
- *Remove Track* — deletes a track entry from the database.
- *Hostile Page* — returns a block (page) of several track entries. Multiple invocations can be made to retrieve all of the hostile entries in the database. This operation is used to update the Operator Console display.
- *Friendly Page* — is equivalent to *HostilePage* except it returns entries for

friendly tracks.

- **Find Best Fit** — returns the `track_id` of the track that correlates most closely with a given position. A special indication is returned if there are no matching tracks. Correlation is based on available IFF information and predicted aircraft positions. This operation is used by the plot correlation thread to match radar reports with existing tracks.
- **Track Hostiles** — is a specialized operation used by the `SAMController` object to locate hostile tracks that will soon come within range of a SAM defense site.
- **Age** — scans the database for tracks that have not been updated recently. Track entries that are too old (because the aircraft was destroyed or moved out of range) are deleted. A thread is started in this object by `Initialize`.

6.2 Mission Planning Objects

6.2.1 Track Handler



The *TrackHandler* object coordinates the response to new tracks that enter the system. The *ProcessTrack* operation is invoked for every new track. It is primarily responsible for establishing the identity of unknown tracks. When hostile tracks are discovered, *ProcessThreat* uses the other mission planning objects to assess threat potential, choose weapons, allocate weapons, and initiate weapons launch. *ProcessThreat* encapsulates the tactical decision functions that determine how the system responds to threats.

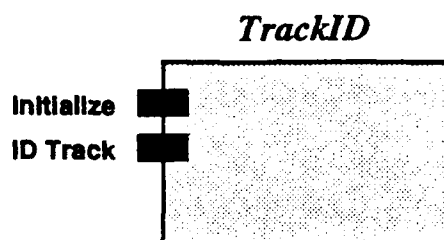
The operations defined on the *TrackHandler* are:

- *Initialize* — initializes internal data structures and creates a thread in the *Dispatch* operation.
- *Enter Track* — registers the existence of a new track in an internal queue. This operation is invoked by `PlotCorrelator.CorrelateFrame` when a new track is discovered.
- *Dispatch* — creates threads to handle the processing of new tracks. A *Dispatch* thread loops in this operation, removing entries from the new track queue (see *Enter Track*) and creating threads in the *ProcessTrack* operation.
- *Process Track* — handles the processing of a new track that has entered the

system. The track is identified using the *TrackID* object and, if hostile, is handed off to the *ProcessThreat* operation. This operation registers each new track with the *WeaponsManager* object so that the activity can be recovered if a node fails.

- *Process Threat* — determines and implements the proper response to a new threat. The *ThreatAssess* and *WeaponsManager* objects are used to determine the threat potential and to choose and allocate an appropriate weapon. Once a weapon has been chosen, a new thread is created to launch and guide the weapon to its target.
- *Recover* — restarts activities that were lost when a node failed. These activities include new track processing and weapons guidance. Section 8.1 describes node failure recovery in greater detail.

6.2.2 Track Identifier

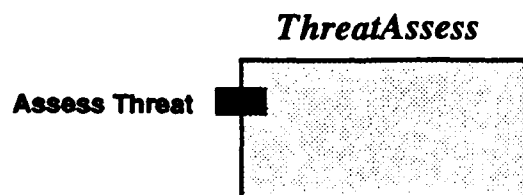


The track identifier (*TrackID*) object determines the identity of a track based on origin, velocity, and IFF information. The initial version of this object relies on very simple heuristics to determine the identity—friendly aircraft are assumed to transmit reliable IFF information, while other aircraft are assumed to be hostile. Velocity and altitude are used to discriminate between bombers, missiles, and interceptors.

The operations defined on *TrackID* are:

- *Initialize* — initializes internal state.
- *ID Track* — identifies a track based on altitude, velocity, and IFF information.

6.2.3 Threat Assessment

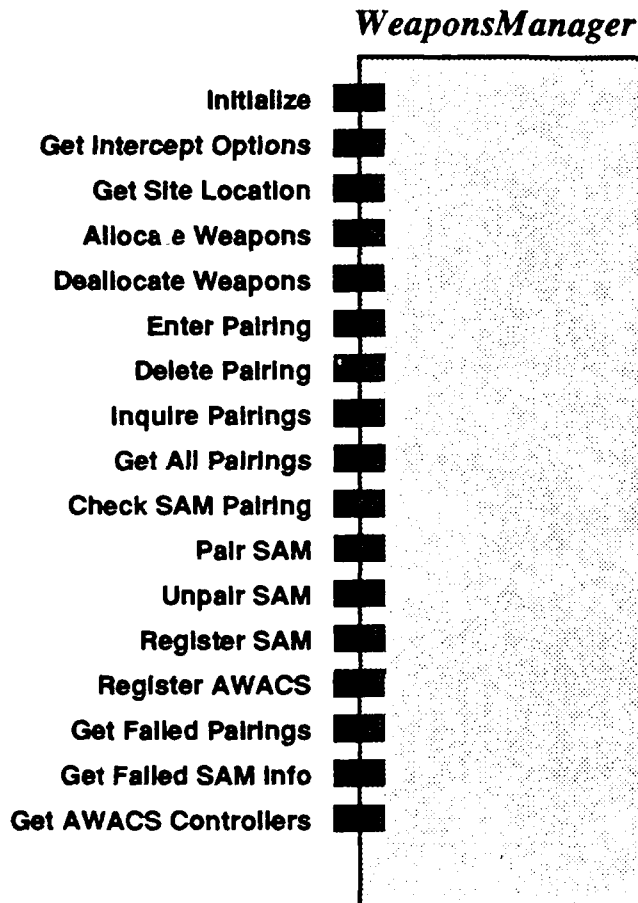


The threat assessment (*ThreatAssess*) object calculates the threat potential of a track based on its position, velocity, and type. Tracks become more threatening as their estimated travel time to the coast decreases. This object is used by the *TrackHandler* and weapon control objects to assign importance values to time constraints.

The only operation define on *ThreatAssess* is:

- *Assess Threat* — assesses the threat potential of a hostile track.

6.2.4 Weapons Manager



The *WeaponsManager* object maintains a database of interceptors and missiles that are available at each airbase. The *TrackHandler* consults this object to identify the type and location of weapons that can intercept a particular threat. Once the *TrackHandler* has chosen a weapon, it invokes the *WeaponsManager* to allocate it. Weapons that have been allocated are removed from the pool of available resources. Guided missiles are removed permanently, while manned interceptors are returned to the pool after they have completed their mission and returned to base.

The *WeaponsManager* also maintains a list of active target/weapon pairings. This pairing information is transmitted to the Operator Console where it is (optionally) used to display pairing lines and predicted intercept information. The pairing information is also used after a node failure to restart weapon guidance threads that were executing on the failed node.

The operations defined on the *WeaponsManager* are:

- *Initialize* — initializes internal data structures, including the weapons database.

- *Get Intercept Options* — computes the predicted time required to intercept a given threat for each available weapon type. The options are returned in order of increasing intercept times.
- *Get Site Location* — returns the coordinates of an airbase based on its site ID.
- *Allocate Weapons* — allocates a particular weapon (airborne interceptor or guided missile).
- *Deallocate Weapons* — deallocates a particular weapon (e.g., after an interceptor has returned to base after a mission).
- *Enter Pairing* — registers a pairing that associates a hostile track with a particular weapon (either a guided missile or manned interceptor). The pairing information specifies the node where the weapon guidance thread is running.
- *Delete Pairing* — removes a target/weapon pairing from the database.
- *Inquire Pairings* — checks to see if one or more weapons are already paired with a particular hostile track.
- *Get All Pairings* — returns a list of all target/weapon pairings
- *Check SAM Pairing* — checks to see if a SAM site has been paired against a particular hostile track.
- *Pair SAM* — registers the assignment of a particular SAM site to a hostile track.
- *Unpair SAM* — removes a target/SAM pairing.
- *Register SAM* — saves the name of the node where the *SAMController* is currently active. This information is used to create a new *SAMController* if the previously active one is lost because of a node failure.
- *Register AWACS* — registers an AWACS guidance activity (including node placement). This information is used in the event of a node failure to restart AWACS control threads that were active on the lost node.
- *Get Failed Pairings* — returns a list of pairings registered on a particular node (which is presumed to have failed). The pairings are also removed from the pairing database.
- *Get Failed SAM Info* — checks to see if the *SAMController* object was active on a lost node.
- *Get AWACS Controllers* — returns a list of AWACS guidance threads that were active on a failed node.

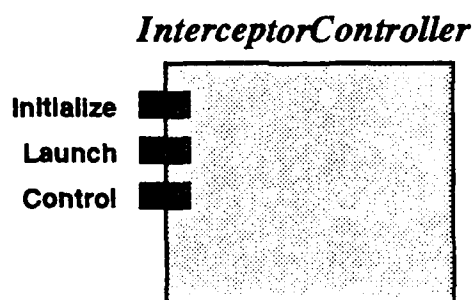
6.3 Weapon Systems Control Objects

The weapon systems control objects are responsible for launching individual weapons and for providing necessary in-flight guidance. The primary weapon systems are manned interceptors and guided missiles. The secondary systems are terminal defense SAMs and airborne early warning aircraft (AWACS).

Interceptors and missiles require frequent course corrections to intercept their targets successfully. These guidance updates are required in part because threats may change course and in part because the weapon controllers use *pursuit guidance* algorithms. Pur-

suit guidance uses the current position of a target to compute a weapon's course. An alternate guidance algorithm, *lead guidance*, predicts the future position of a target by extrapolating its current flight path. Lead guidance would require fewer course corrections and would likely result in quicker intercepts. However, pursuit guidance has a particular advantage for the ADSP. One of the goals of the application effort was to demonstrate the effects of proper time-driven scheduling. If lead guidance were used, weapons could hit their target with only a single guidance command (assuming the target did not change course or speed). Using pursuit guidance ensures that there will be a visible effect when guidance updates are delayed. This allows observers to determine when load shedding occurs and to see which threats have been bypassed.

6.3.1 Interceptor Controller

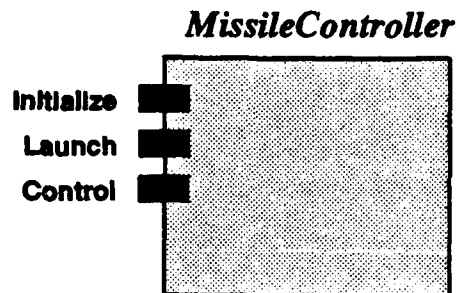


The *InterceptorController* object is responsible for launching manned interceptors and guiding them to their targets. The *Launch* operation transmits a launch request to the Scenario Simulator and registers the target/weapon pairing with the *WeaponsManager*. Once an interceptor has been launched, control is passed to the *Control* operation which calculates further course updates. The *Control* operation also determines when the next course update should be made (based on the estimated intercept time) and establishes the proper time constraints. When the threat has been intercepted, the *Control* operation guides the interceptor back to base where it lands and is returned to the pool of available weapons.

The operations defined on the *InterceptorController* are:

- *Initialize* — initializes internal data structures.
- *Launch* — launches and initiates control of a particular interceptor. Launching consists of computing an initial trajectory, creating a track for the interceptor in the *TrackDB*, and issuing the launch command to the Scenario Simulator. The newly launched weapon is registered with the *WeaponsManager* and control is passed to the *Control* operation.
- *Control* — steers a weapon to its target by issuing intermittent guidance updates. The frequency of the guidance updates is determined by the distance between the interceptor and its target. Guidance updates are computed using a pursuit guidance algorithm.

6.3.2 Missile Controller

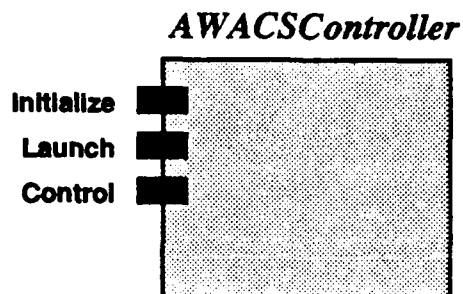


The *MissileController* object is responsible for launching missiles and for guiding them until they intercept their targets. The *MissileController* is very similar to the *InterceptorController*. The major difference between the two object types is the handling of "kills." It is possible for a missile to "hit" the wrong target (and be destroyed) by passing too close to it. If the missile is destroyed and its original target still exists, the guidance activity notifies the *TrackHandler* to allocate and launch a new weapon against the hostile track. Similar situations can occur when a missile's target is killed by another weapon. If the guidance thread is unable to locate the threat, a self-destruct command is issued to the Scenario Simulator to remove the missile.

The operations defined on the *MissileController* are:

- *Initialize* — initializes internal data structures.
- *Launch* — launches and initiates control of a particular missile. Launching consists of computing an initial trajectory, creating a track for the missile in the *TrackDB*, and issuing the launch command to the Scenario Simulator. The newly launched weapon is registered with the *WeaponsManager* and control is passed to the *Control* operation.
- *Control* — steers a weapon to its target by issuing intermittent guidance updates. The frequency of the guidance updates is determined by the distance between the missile and its target. Guidance updates are computed using a pursuit guidance algorithm.

6.3.3 AWACS Controller

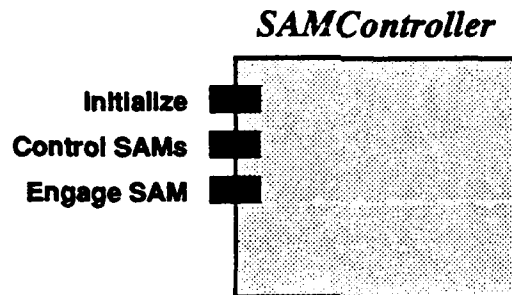


The *AWACSController* object guides an airborne early warning and control aircraft in an oval pattern over its coverage area. The coverage area for each AWACS is defined by the human operator. This placement information is passed to the *Launch* operation which sends a launch command to the Scenario Simulator and calculates the initial guidance information. Each AWACS launch is registered with the *WeaponsManager* so that the guidance activity can be restarted if it is lost because of a node failure. Control of the aircraft is then passed to the *Control* operation which calculates the course corrections necessary to maintain the oval. AWACS planes do not require course updates while flying the straight legs of their flight pattern. However, since guidance commands are issued in terms of absolute headings, they require frequent updates while navigating the circular portions of their course.

The operations defined on the *AWACSController* are:

- *Initialize* — initializes internal data structures.
- *Launch* — launches and initiates control of an early warning aircraft. Launching consists of computing an initial trajectory, creating a track for the AWACS in the *TrackDB*, and issuing the launch command to the Scenario Simulator. The newly launched AWACS is registered with the *WeaponsManager* and control is passed to the *Control* operation.
- *Control* — steers an AWACS in an oval pattern by issuing intermittent guidance updates. No course updates are required while the aircraft is flying the straight legs of the pattern. Frequent updates are required to navigate the turns at the end of the pattern.

6.3.4 SAM Controller

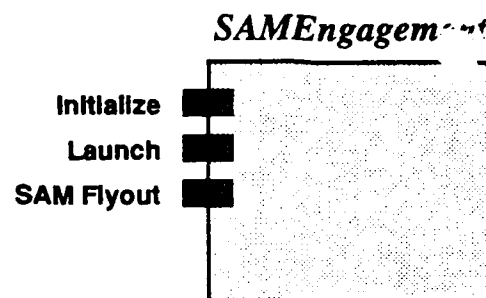


The terminal defense SAM batteries are managed by two objects—the *SAMController* object and the *SAMEngagement* object. The *SAMController* is responsible for detecting hostile tracks that are about to enter the range of the SAM batteries. When hostile tracks are detected within range, the *SAMController* engages the appropriate SAM by starting a thread in the *SAMEngagementLaunch* operation.

The operations defined on *SAMController* are:

- *Initialize* — initializes internal data structures and creates a thread in the *ControlSAMs* operation.
- *Control SAMs* — monitors the locations of hostile tracks to identify threats that will soon enter the range of a SAM battery.
- *Engage SAM* — initiates the *SAMEngagementLaunch* operation.

6.3.5 SAM Engagement



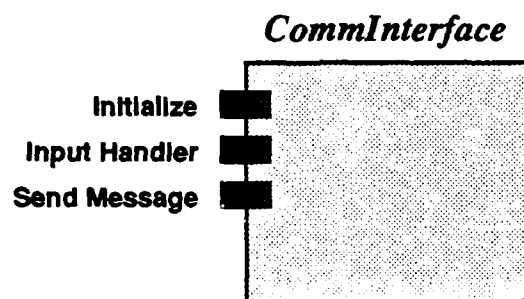
The *SAMEngagement* object simulates the aim-aim-fire sequence of a SAM battery using a sequence of delays and deadlines. The *Launch* operation assigns a particular SAM to a hostile craft and waits for the target to come within range. When the target is within reach, the *SAMFlyout* operation is invoked to simulate the aiming and firing sequence. If the *Launch* and *SAMFlyout* operations successfully meet their time constraints, the SAM is defined to be successful and a kill signal is sent to the Scenario Simulator. If any of the time constraints are not satisfied, the launch operation is retried. This sequence continues until the target is either killed or travels out of range of the SAM battery.

The operations defined on *SAMEngagement* are:

- *Initialize* — initializes internal data structures.
- *Launch* — assigns a particular SAM to a threat and monitors the target distance to the kill ring. When the target is in range, the *SAMFlyout* operation is invoked to engage the weapon.
- *SAM Flyout* — simulates the aim-aim-fire sequence of a SAM launch. Each stage of the sequence is simulated by a time-constrained delay.

6.4 System Support Objects

6.4.1 Communication Interface

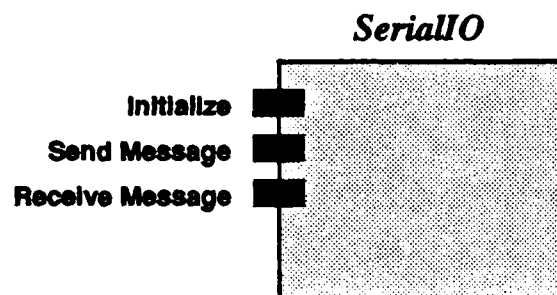


The *CommInterface* object manages communication between the ADSP (running under Alpha) and the simulated environment (running under UNIX). The actual message transmission is handled by either the *NetworkIO* or *SerialIO* objects. A thread in the *InputHandler* operation accepts messages as they arrive from UNIX and invokes operations on the appropriate destination objects to deliver the incoming data. When an Alpha thread needs to send a message to UNIX, it invokes the *SendMessage* operation with the message to be transmitted.

The operations defined on *CommInterface* are:

- *Initialize* — selects and initializes the communications link (either *NetworkIO* or *SerialIO*), creates a thread in the *InputHandler* operation to dispatch incoming messages, and initializes internal data structures.
- *Input Handler* — receives messages from the communication link and invokes operations on the appropriate objects based on the message type.
- *Send Message* — transmits an outbound message (from Alpha to UNIX) over the active communication link.

6.4.2 Serial I/O

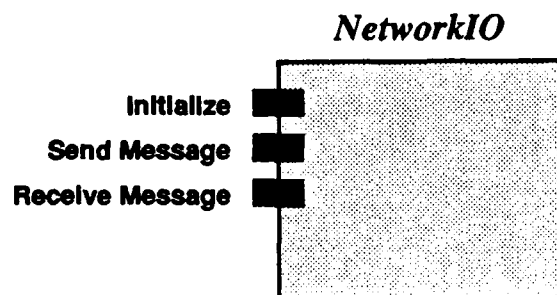


The *SerialIO* object provides a reliable communications link between an Alpha node and a UNIX host over an RS422 serial line. *SerialIO* uses a custom protocol to ensure the accurate reception of data packets. The protocol attaches a checksum and message number to each message sent across the link. When a message is received, the checksum is re-computed and compared with the transmitted value. If the checksums disagree, a negative acknowledgment (NACK) for that message is returned. If the checksums agree, no response is generated (implicit acknowledgment). When the sender receives a NACK, it re-transmits the corrupted message and all messages sent after the corrupted one.

The operations defined on *SerialIO* are:

- *Initialize* — initializes the serial line driver and protocol buffers and establishes communication with the UNIX side of the communication link.
- *Send Message* — transmits an outbound message (from Alpha to UNIX) over the serial line using the reliable serial protocol.
- *Receive Message* — reads the next message from the serial line and returns it in the caller's message buffer. The invoking thread is blocked until a message is available.

6.4.3 Network I/O



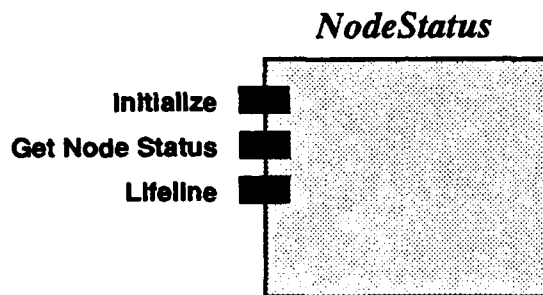
The *NetworkIO* object provides a reliable communications link between an Alpha node and a UNIX host using an Ethernet-based communication protocol. The *NetworkIO* object acts as an interface to the Communications Processor (CP) on the Alpha node. The CP is responsible for transmitting and receiving messages using the Alpha external com-

munications (EXT) protocol. On the UNIX side of the connection, a special daemon handles the reception and transmission of EXT messages.

The operations defined on *NetworkIO* are:

- *Initialize* — initializes the message queue and establishes communication with the UNIX side of the communication link.
- *Send Message* — transmits an outbound message using the EXT protocol.
- *Receive Message* — reads the next message from the Communication Processor and returns it in the caller's message buffer. The invoking thread is blocked until a message is available.

6.4.4 Node Status

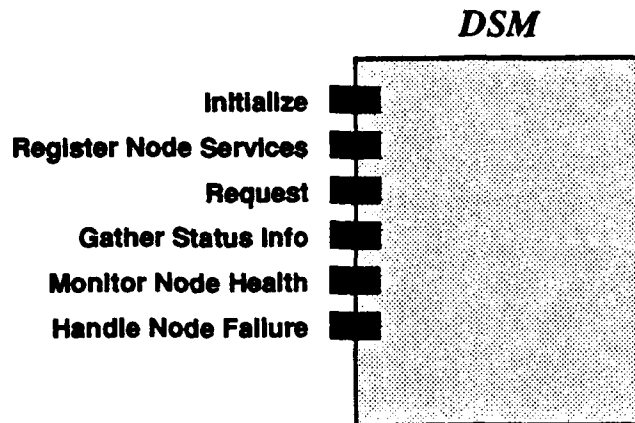


The *NodeStatus* object provides load and performance statistics for the node on which it resides. The *GetNodeStatus* operation returns information about such things as memory utilization, number of object instances, number of active threads, and scheduling queue lengths. The *Lifeline* operation is used by the *RecoveryManager* object to detect when a node has failed.

The operations defined on *NodeStatus* are:

- *Initialize* — initializes the statistics gathering buffers.
- *Get Node Status* — returns information about such things as memory utilization, number of object instances, number of active threads, and scheduling queue lengths for the local node.
- *Lifeline* — blocks indefinitely on a semaphore. (A thread that invokes this operation should never return to the invoking object. If the thread does return, it means the thread has been broken due to a node failure.)

6.4.5 Distributed Services Manager



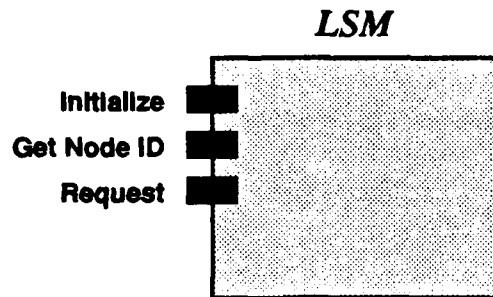
The distributed services manager (*DSM*) object manages the distribution of objects and threads among different nodes. When the application needs to create a new object or thread, it *Requests* the new service from the *DSM*. The *DSM* then places the new object or thread on the most lightly-loaded node. The actual creation of objects and threads is handled by a local services manager (*LSM*) on the target node. (This scheme of using the *LSM* as an agent was a convenient shortcut to ensure that threads would be rooted on the desired node. A more general solution is expected to be included as a mechanism in a later Alpha release.)

Load information is maintained by a thread that periodically requests status updates from the application nodes. The *DSM* is also responsible for initiating recovery operations in the event of a node failure. The *MonitorNodeHealth* operation sends “feeler” threads out to the different application nodes and initiates the *HandleNodeFailure* operation when any of the application nodes fail.

The operations defined on *DSM* are:

- *Initialize* — initializes local data structures and starts the status gathering and monitoring functions.
- *Register Node Services* — registers the *LSM* and *NodeStatus* objects for a node. The newly registered node is assigned a logical ID that is returned as a parameter.
- *Request* — invokes *LSM.Request* on the most lightly-loaded node in the system.
- *Gather Status Info* — periodically updates the status and loading information for each of the application nodes.
- *Monitor Node Health* — sends a “feeler” thread to the *NodeStatus.Lifeline* operation on a particular node.
- *Handle Node Failure* — marks a node as down and initiates a recovery operation on the most lightly-loaded node.

6.4.6 Local Services Manager

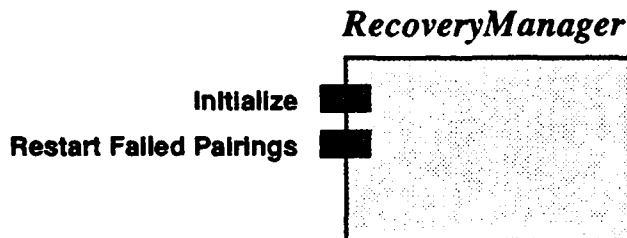


The local services manager (*LSM*) object acts as an agent of the *DSM*—creating local objects and threads on its behalf. The *LSM* is also responsible for registering the local node with the *DSM* when first powered up. When the node registers, it is given a logical node ID that is used by the application to track the location of objects and thread roots in case the node fails.

The operations defined on *LSM* are:

- *Initialize* — registers the local node with the *DSM* and initializes the logical ID of the local node.
- *Get Node ID* — returns the logical ID of the local node.
- *Request* — creates a local object or thread on behalf of the *DSM*.

6.4.7 Recovery Manager



The *RecoveryManager* object is invoked by the *DSM* when a node failure has been detected. The *RecoveryManager* is responsible for starting new instances of weapon (AWACS, Interceptor, Guided Missile, and SAM) control threads that were active on the failed node.

The operations defined on *RecoveryManager* are:

- *Initialize* — initializes internal state.
- *Restart Failed Pairings* — restarts controller threads that were active on a node that has failed.

7 Air Defense System Threads

The air defense system object types define the functions that can be performed by the application. The system threads define the sequence of activities that are actually executed.

The ADSP threads were defined by examining the functional decomposition given in Section 5.2. Threads were initially chosen by locating logically-related sequences of activities. Once the sequences had been refined, decisions were made about when a single thread should perform several jobs in succession and when separate thread instances should be created to execute concurrently. The resulting design defines several different types of threads. In some cases there is a single instance of a particular thread type in the entire system. For other types, separate threads are created to handle each track or each weapon that is active.

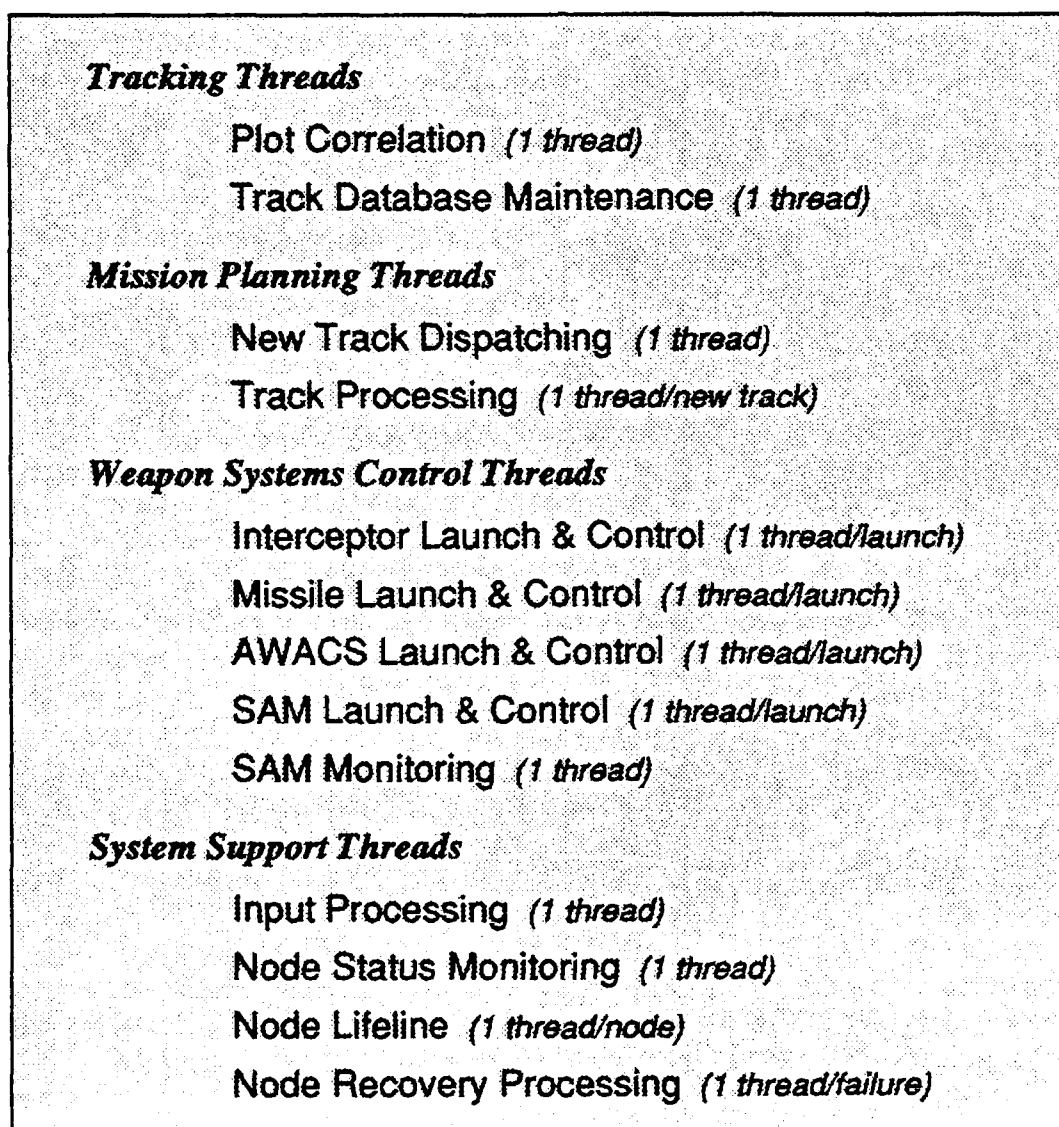


Figure 11: Air Defense System Threads

Figure 11 lists the different types of threads in the ADSP and shows how many instances of each are created. The remainder of this chapter describes each of the thread types in greater detail.

7.1 Tracking Threads

7.1.1 Plot Correlation

The plot correlation thread is responsible for updating the tracking database based on incoming radar reports. A single instance of the thread is rooted in the *PlotCorrelator.CorrelateFrame* operation. When a new frame of data has been received, the thread retrieves the next timestamp from the input buffer (see *PlotCorrelator.GetStamp*) and begins processing the radar reports. For each report, the thread correlates the location of the report with the contents of the tracking database (see *TrackDB.FindBestFit*) and updates the database as necessary. When new tracks are discovered, the thread registers their existence using the *TrackHandler.EnterTrack* operation so that a new track processing thread can be created. When an entire frame has been correlated, a new display list is transmitted to the Operator Console using the *CommInterface.SendMessage* operation. Finally, the plot correlation thread loops back and waits until the next frame of data is available.

7.1.2 Track Database Maintenance

The track database maintenance thread removes ("ages") old tracks that are no longer active. A single maintenance thread resides in the *TrackDB.Age* operation. Every few seconds (once per radar frame), the thread scans the database for threads that have not been updated recently. Track entries that are too old (because the track moved out of range or was destroyed) are deleted.

7.2 Mission Planning Threads

7.2.1 New Track Dispatching

The new track dispatching thread creates a track processing thread for every new track that enters the system. A single thread rooted in *TrackHandler.Dispatch* waits for new tracks to be identified by the plot correlation thread, and creates a new thread in the *ProcessTrack* operation. Creating a thread for each track allows mission planning operations such as track identification and weapon assignment to be carried out concurrently for several tracks, and allows the time constraints for each of the threads to be set independently according to the threat potential of the associated track.

7.2.2 Track Processing

Track processing threads perform mission planning activities associated with new tracks. First, the *TrackID* object is invoked to determine whether the new track is friendly or hostile. If the track appears to be hostile, the track processing thread passes into the *TrackHandler.ProcessThreat* operation where functions such as threat assessment and weapon assignment are performed. The final act of the track processing thread is to launch any missiles or interceptors that are assigned to intercept the track. Once the weapons have been launched, the thread dies.

7.3 Weapon Systems Control Threads

7.3.1 Interceptor Launch and Control

The interceptor launch and control (L&C) thread launches an individual interceptor and provides any necessary in-flight guidance to ensure an accurate intercept with the target. A separate L&C thread is created for every interceptor launched. Each thread is started in the *InterceptorController.Launch* operation where it issues the launch request and registers the pairing with the *WeaponsManager*. The thread then invokes the *Control* operation to guide the interceptor to its target. The L&C thread is responsible for detecting when the target has been successfully intercepted and for guiding the interceptor back to its base after the completion of the mission. When the interceptor is safely home the thread dies.

Since a separate L&C thread is created for every interceptor, the system can independently set the time constraints of each weapon control activity based on the potential threat of its target. The exact nature of these time constraints is described in Sections 3.3.2 and 8.2.1.

7.3.2 Missile Launch and Control

The missile L&C threads are identical to the interceptor L&C threads except that missiles that survive their mission (because another weapon intercepted their target) are issued self-destruct commands rather than being guided back to their base.

7.3.3 AWACS Launch and Control

An AWACS L&C thread is responsible for launching a single AWACS aircraft and guiding it along an oval-shaped flight path. A separate thread controls each of the early warning aircraft. Each L&C thread adjusts its time constraints dynamically depending on whether the controlled aircraft is flying along a straight leg of the flight path (where course adjustments are not required) or is flying around an end curve (where adjustments must be made frequently).

7.3.4 SAM Launch and Control

SAM L&C threads manage the targeting and firing of individual SAM batteries. When the SAM monitoring thread detects a hostile target about to enter the range of a SAM battery, it creates a SAM L&C thread in the *SAMEngagement.Launch* operation. The L&C thread waits for the target to come within range, then invokes the *SAMFlyout* operation. The thread then enters a series of time constraints designed to simulate an aim-aim-fire operation (see Sections 3.3.3 and 8.2.1). If the time constraints are satisfied, the launch is defined to be successful and a kill message is issued to the Scenario Simulator. (This is the only instance where the ADSP system running under Alpha issues a kill command. In all other cases, kills are detected by the Scenario Simulator and must be discovered by the ADSP. The short time scale of SAM engagements made this small aberration necessary.)

7.3.5 SAM Monitoring

The SAM monitoring thread is responsible for detecting hostile tracks that are nearing the range of the terminal defense SAMs. A single monitoring thread is created at system

initialization time. When a target is discovered (using *TrackDB.TrackHostiles*), the SAM monitoring thread assigns a SAM battery and initiates the engagement by creating a SAM L&C thread in the *SAMEngagement.Launch* operation.

7.4 System Support Threads

7.4.1 Input Processing

The input processing thread collects messages from the Communication Interface and distributes them to the appropriate objects based on the message type. The majority of messages received are radar plot reports which the input thread stores using the *PlotCorrelator.BufferReport* operation. If the *PlotCorrelator* object does not exist (e.g., because of a node failure), the input processing thread requests the creation of a new correlator (see Section 8.1).

7.4.2 Node Status Monitoring

A single node status monitoring thread travels around the system gathering node status information (e.g., number of active threads, memory utilization, etc.) for each node. This information is used by the *DSM* to direct load balancing and to display monitoring information on the Experimental Control Console. The status monitoring thread is rooted in *DSM.GatherStatusInfo* and travels between the *NodeStatus.GetNodeStatus* operations on each node.

7.4.3 Node Lifeline

A lifeline thread is created for each node when it registers itself at system initialization or after a node recovery. The thread is created in *DSM.MonitorNodeHealth* where it immediately invokes *NodeStatus.Lifeline* on the destination node. If a monitored node fails, Alpha reports a thread break exception to the *MonitorNodeHealth* operation and the lifeline thread invokes *DSM.HandleNodeFailure* to initiate recovery processing. A more detailed description of the failure recovery mechanism is given in Section 8.1.

7.4.4 Node Recovery Processing

When a node fails, any weapons L&C threads that were active on that node must be restarted on the remaining nodes. The *DSM.HandleNodeFailure* operation creates a node recovery processing thread in the *RecoveryMgr.RestartFailedPairings* operation. The recovery thread retrieves a list of the lost threads using the *GetFailedPairings* and *GetFailedSAMInfo* operations of the *WeaponsManager*. It then creates new L&C threads for each of the pairings using the *DSM.Request* operation. When all of the threads have been regenerated on other nodes, the recovery processing thread dies.

8 Technology Evaluation

The primary goals of the Alpha application effort were to apply the decentralized, real-time technology incorporated in the Alpha operating system, and to evaluate that technology in a realistic context. This chapter discusses and evaluates how the advanced systems technology provided by Alpha was used to meet the distribution, survivability, and timeliness requirements of the air defense application.

8.1 Distribution and Survivability

Distributed computers are rapidly becoming a major component in the design of large application systems. Alpha is designed to integrate these physically dispersed nodes into a single decentralized computer system—providing transparent access to, and unified management of, distributed resources. This integration allows applications to obtain the many performance, availability and survivability benefits of distributed systems without incurring many of the difficulties often associated with conventional distributed programming.

The air defense system prototype is designed to operate, without modification, either on a single centralized computer or on a network of distributed computing nodes. The actual system has been tested in configurations containing between one and four Alpha nodes. When executing in a distributed environment, the ADSP automatically distributes work among the available processors. If one or more of the nodes fail, the ADSP automatically recovers and continues to operate using the remaining nodes. If new or repaired nodes become available, they are seamlessly integrated into the already running system.

To achieve this level of availability and survivability, the design team at GD (working closely with the Alpha group at CMU) developed a plan for distributing the ADSP objects and threads among the available nodes. Figure 12 illustrates a sample distribution of objects and threads between two of the application nodes. The bar graph at the left of each display monitors the total number of threads active on the node. The listing on the right

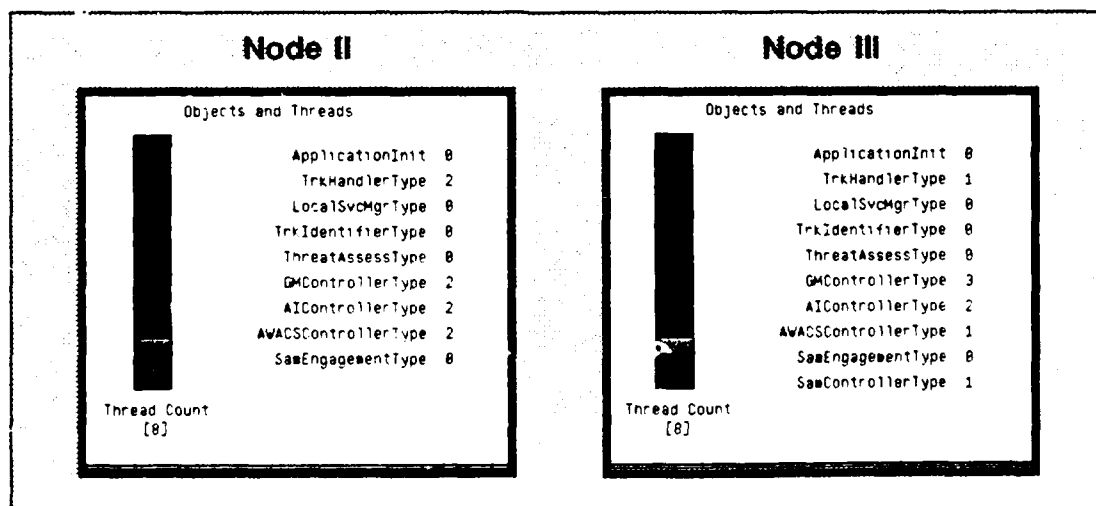


Figure 12: Typical Distribution of Objects and Threads

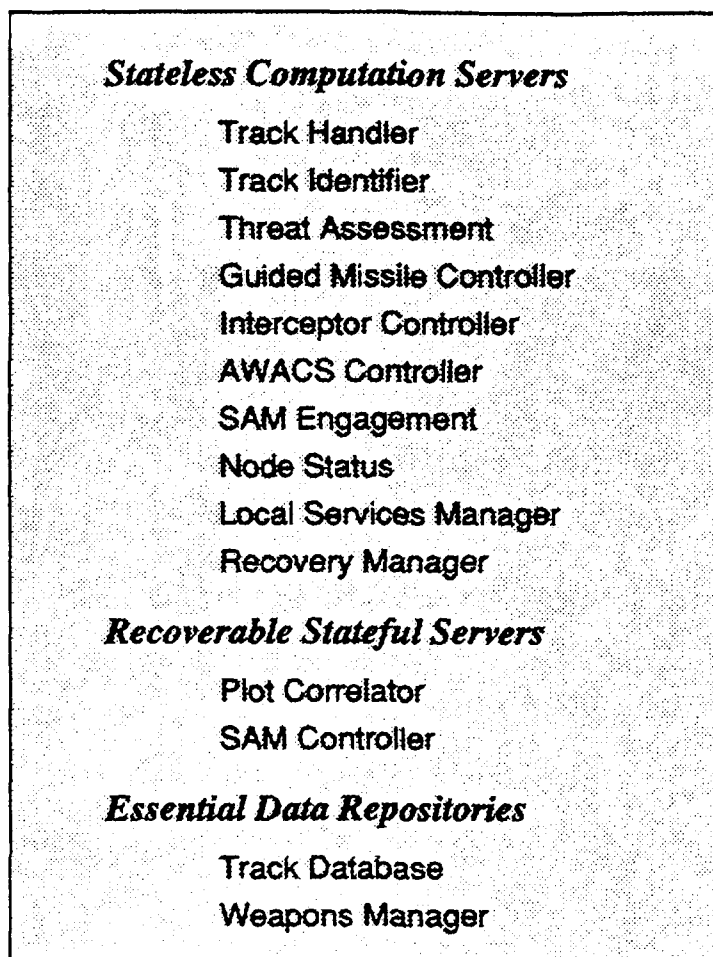


Figure 13: Object Distribution Classifications

shows the object instances present on the node and the number of threads active in each object. (The object names listed in Figure 12 differ slightly from those used in this report. The expanded names used in the report correspond directly to the object types used in the ADSP, but are more legible.) The remainder of this section describes in greater detail how the objects and threads are distributed and how that distribution allows the ADSP to recover and continue operating after node failures.

8.1.1 Object Distribution

Since Alpha provides for the transparent distribution of objects among processing nodes, applications can largely be designed as if they were going to execute on a single processor. Distribution issues become important when considering the survivability and availability requirements of the application. In particular, the survivability requirements determine which objects need to be inclusively replicated, while availability considerations influence the number and placement of object instances.

The survivability requirements of the air defense application were outlined in Section 3.4. In designing the ADSP, those requirements were used to classify the application objects into three categories—stateless computation servers, recoverable stateful servers,

and essential data repositories. The categories are primarily distinguished by the amount and type of application data maintained in the objects. Figure 13 lists the object types in each class. The remainder of this section outlines their characteristics.

The stateless computation server (SCS) objects do not maintain any important state information between incoming operation invocations (*i.e.*, they contain executable code, but no important local data). As a result, if one of the SCS objects disappears because of a node failure, no vital information is lost. Independent instances of the SCS objects can be maintained concurrently on several nodes since no application data must be kept consistent between the copies. For the ADSP, the designers decided to create local instances of every SCS object on each of the application nodes. Maintaining local copies on each node reduces the number of remote invocations required and simplifies failure recovery by eliminating the need to create replacement objects after a node failure.

Unlike the SCS objects, the recoverable stateful server (RSS) objects do maintain application state information between invocations. The information they store, however, is often transient and can either be recreated or replaced if it is lost because of a node failure. The best example of an RSS object is the *PlotCorrelator*. The *PlotCorrelator* buffers radar reports until they are processed. If the *PlotCorrelator* is lost, the radar reports that have been buffered will be lost also. However, radar reports are constantly being generated by the sensors. Under the stressful conditions of a node failure, the ADSP designers decided that it is better to process current reports as they arrive than to use potentially limited processor cycles to restore and process outdated information. At most one instance of any RSS object exists at any particular time. If that instance disappears because of a node failure, a new instance is created on demand on one of the remaining nodes.

The essential data repository (EDR) objects maintain information that is vital to the continued operation of the ADSP. The tracking database and weapons resource database are the primary examples. The data stored in EDR objects should remain available and consistent as long as any of the application nodes are functioning. To achieve this level of reliability, the EDR objects should be inclusively replicated on multiple application nodes. Since multiple copies of replicated objects are maintained in a consistent state, the loss of a single node would not result in the loss of the data. The number of replicas maintained can be set depending on the level of reliability desired.

Since the Alpha replication facilities were not in place in time for the ADSP demonstration, single instances of the EDR objects were placed on a system interface node along with the communications functions. The need for an interface node originated because of the serial communication link between the ADSP and the experimental environment running under UNIX. We did not have the time or the desire to implement redundant communication links, and so decided to dedicate one node to act as the interface and to hold (temporarily) the EDR objects. The remainder of the nodes are interchangeable "application" nodes that handle the bulk of the application processing.

8.1.2 Thread Distribution

In addition to survivability, one advantage of distributed systems is the ability to execute several activities concurrently. In the ADSP, multiple instances of several thread types can be active at any time. While one node is processing a new frame of radar data,

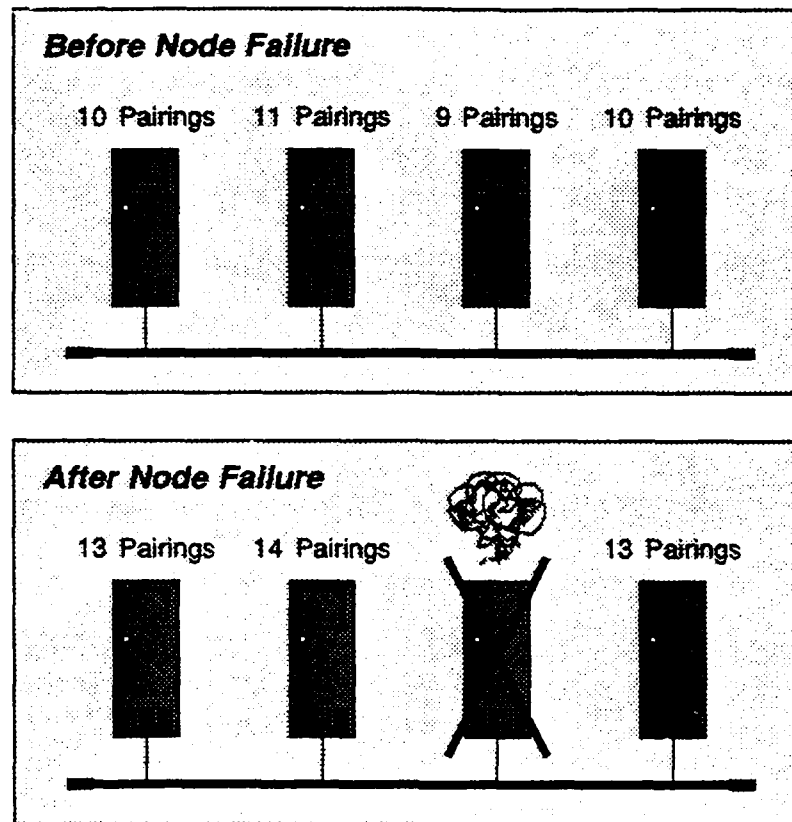


Figure 14: Reconfiguration After a Node Failure

other nodes may be computing guidance information for weapons that have already be launched. In general, it is desirable to distribute the processing load evenly among the nodes (to avoid local overloads). Load balancing methods for distributed system are the topic of continuing research. For the ADSP, load balancing is managed by the *DSM*. The *DSM* maintains status information on each of the nodes in the system. It uses this status information (including number of active threads, memory utilization, *etc.*) to estimate the load on each node. Each time a new mission planning or weapon systems control thread is required, an invocation is made on the *DSM* to create the new activity. The *DSM* then forwards the request to the *LSM* on a lightly-loaded node where the thread is then created.

This load balancing scheme works relatively well for the ADSP. Eventually, however, many of the services provided by the *DSM* will be supplanted by more general system services offered by Alpha. Our experience with the *DSM* has given us a better understanding of the types of services that should be offered and the ways in which applications are likely to use them.

8.1.3 Node Failure Response

One of the fundamental survivability requirements specified by the application design team was that the ADSP should continue to operate correctly after one or more node failures. This capability requires both that essential application data remain available and that partially-completed activities active on the failed node be cleaned up and, if neces-

sary, restarted. The designers ensured that important data would remain available after a failure by keeping vital application information in the essential data repository objects. The remaining task of cleaning up and restarting failed activities is handled by Alpha and by the *RecoveryManager*.

The distribution and exception-handling facilities provided by Alpha already handle a significant amount of the processing required when a node fails. Alpha automatically detects a node failure and initiates system cleanup operations. Any threads that were active on the failed node are trimmed and an exception is returned to the invoking object. Exception handling code then initiates application-level recovery operations.

Once the system-level cleanup has been completed, a node failure is signalled to the *DSM* (via the node lifeline thread for the failed node). The *DSM* removes the failed node from the resource pool and starts a new thread in the *RecoveryManager* object. The recovery operation queries the *WeaponsManager* for a list of guidance threads that were active on the failed node. Replacement activities are then created and distributed across the remaining nodes in the system. Any recoverable stateful server objects that were active on the failed node are recreated on demand on one of the remaining nodes. Figure 14 illustrates how weapons guidance threads are redistributed in response to a node failure. In this example, the nine target/weapon pairings being controlled by the third node are re-assigned to the other available nodes.

During public ADSP demonstrations, node failures are simulated by turning off the power to one of the application nodes. By monitoring the node displays on the Experimental Control Console, it is possible to watch the defense system recover by restarting the lost guidance activities on the remaining nodes. The "failed" node is then turned back on and re-integrated into the running defense system.

8.2 Time-Driven Resource Management

The primary requirement of the air defense system is that it prevent hostile targets from violating U.S. airspace. To meet this requirement, the tracking, mission planning, weapon systems control, and system support activities must be carried out on time. If a SAM launch sequence is completed ten seconds after its target has dropped a bomb on New York City, it has failed just as much as if the launch had never been completed. The situation is further complicated by the diverse characteristics and dynamic nature of the time constraints presented by the ADSP. Applications such as air defense have complex timeliness requirements that are not easily expressed in terms of fixed, periodic, or hard deadlines. For example, the urgency and importance of issuing guidance updates when a target is hundreds of miles away is significantly lower than when the target is about to reach the coastline. Finally, the ADSP must continue operating to preserve the coastal integrity even when insufficient processing resources are available (as may be the case in the event of one or more node failures).

Alpha supports this type of complex and changing real-time application through its use of a unified time-driven resource management mechanism. In Alpha, timeliness requirements are specified dynamically by application threads using time-value functions. Time-value functions allow a variety of time constraints to be specified both concisely and accurately. This information is then used by the best-effort scheduler to allocate processor cycles in a way that maximizes the benefit to the application. Under normal conditions

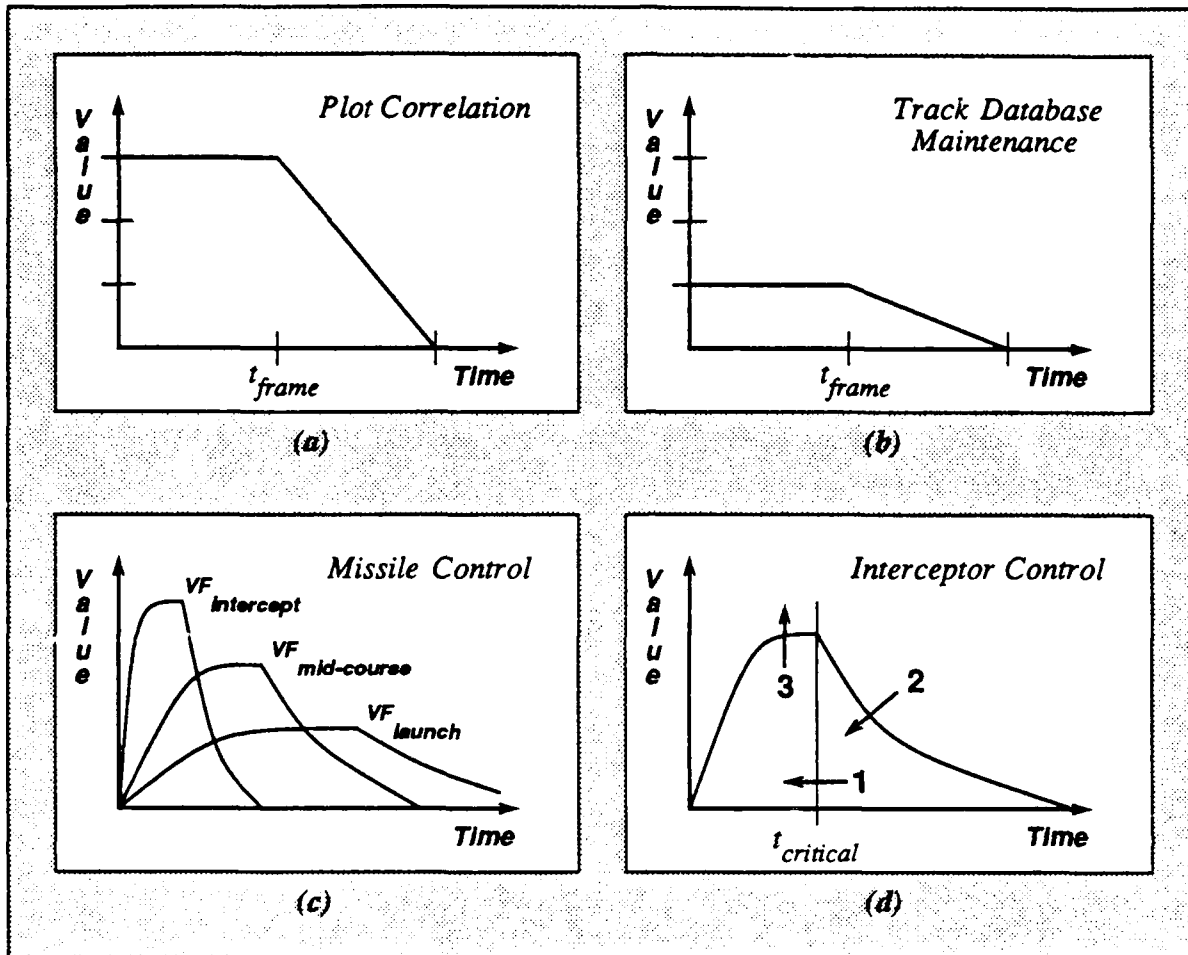


Figure 15: ADSP Time-Value Function Examples

this means that the timeliness requirements of all activities will be satisfied. If the system is overloaded (e.g., because of a node failure), those activities that are the most valuable to the application as a whole will be given preference. For a more complete discussion of Alpha's time-driven resource management mechanisms and policies, refer to Section 2.2.

8.2.1 Application Time-Value Functions

The timeliness requirements of the air defense application were discussed in Section 3.3. Once the application threads had been defined, the designers found that converting these textual requirements into actual time-value functions was fairly straightforward. A significant amount of system "tuning" was eliminated since the team did not have to massage the requirements into a foreign set of real-time specifications such as rate groups or priority levels.

To derive time-value specifications from the written requirements, each of the time-constrained activities was analyzed to determine its critical time and its completion value over time (to determine the shape of the function). The different activities were then compared to determine the amplitude of the function by estimating their relative value to

the system during overload situations. These considerations combined to define the final time-value function specifications.

The plot correlation and track database maintenance threads have critical times corresponding to the radar frame arrival rate. In both cases, it is best if the processing is completed before the next frame of data arrives. However, it is acceptable for the processing to slip as much as one additional time frame under extreme overload situations. The time-value functions for these threads are shown in Figure 15(a) and Figure 15(b). As illustrated, both functions share the same shape and critical time. The plot correlation activity, however, has a much greater value to the system under overload conditions.

The timeliness requirements for the missile and interceptor launch and control threads vary over the course of an engagement. After launch, the guidance control threads must issue timely course updates to ensure a successful intercept. However, the required frequency of these updates and the importance of completing the course corrections at the desired time change both as the distance between the weapon and the target and from the target to the coastline decrease. Figure 15(c) illustrates how the time-value functions for weapon control threads vary over the course of an intercept. The characteristics can be summarized as follows:

- *loose time-constraints* — The critical time indicates the time at which course correction is preferred, not absolutely required.
- *variable critical times* — Course corrections are needed more often as the distance between the threat and the weapon decreases. The desired update time is one fourth of the predicted time to intercept. Arrow 1 in Figure 15(d) illustrates how the critical time changes as the intercept nears.
- *variable "hardness"* — It becomes more important to use the most recent position information as the distance between the threat and weapon decreases. Therefore as the distance narrows, it may be preferable to abort a late update and restart the guidance calculations with fresh information. Arrow 2 in Figure 15(d) shows how this requirement is reflected by a decrease in the completion value of the constraint after the critical time.
- *dynamic maximum* — The value of successfully completing an intercept corresponds to the threat potential of the target being intercepted. The perceived threat depends on changing parameters such as distance from coastline. Arrow 3 in Figure 15(d) illustrates how the time-value function is scaled upward as the threat increases.

Although very different from more conventional real-time specification methods, the ADSP designers were pleased with the expressive ability and ease of deriving time-value functions. Other specification methods would likely not have allowed time constraints for individual activities to be specified so easily. If only one constraint could be specified, all instances of an activity would have to execute under the most stringent constraint of any instance—potentially causing time constraints to be missed unnecessarily. The ability to specify importances, or values, for separate instances of a thread allow the system to make much better resource decisions when the system is overloaded.

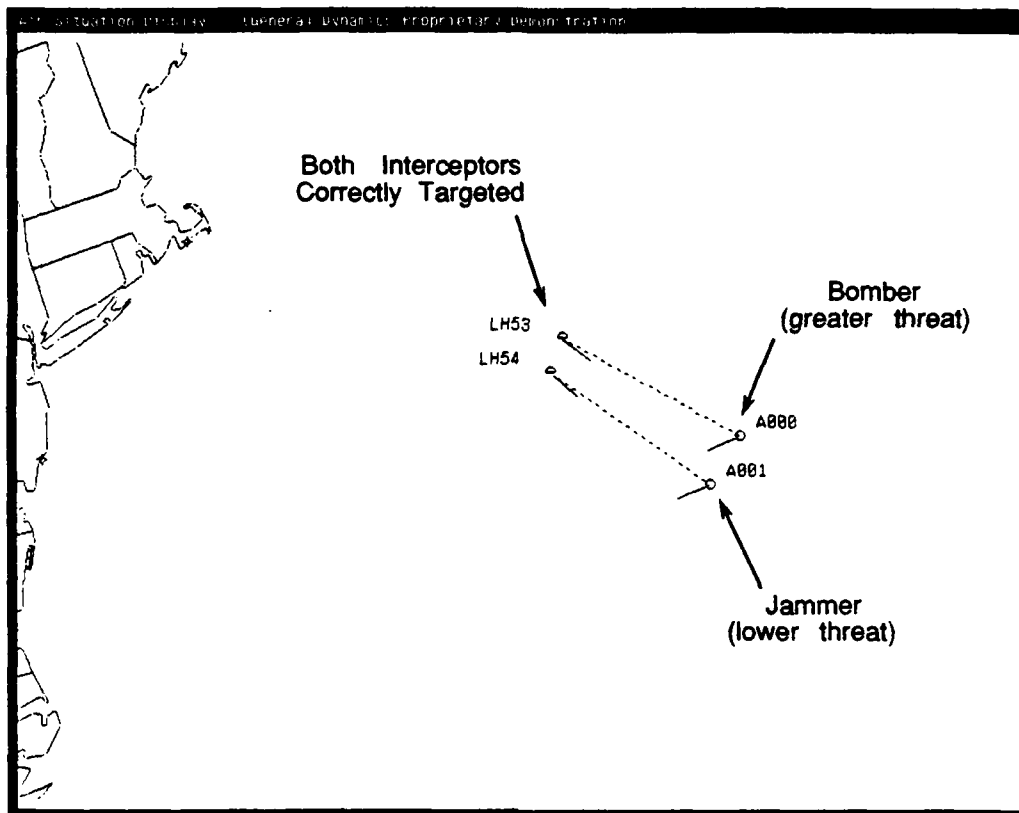


Figure 16: Normal Behavior

8.2.2 System Overload Behavior

Many real-time systems react to temporary overload conditions by breaking completely and permanently. To reduce the probability of such failures occurring, designers may require such massive redundancy that "overloads will never happen." Besides being wasteful, such techniques may not work if a system suffers multiple simultaneous failures (as might be expected during an intentional attack).

The best-effort scheduler used by Alpha responds to overload conditions by concentrating its efforts on completing activities that will be the most valuable to the application (as specified by the time-value functions). For the ADSP, this means that the Operator Console display is accurately maintained and that targets within the range of the SAM defenses receive primary attention. Other targets receive attention based on their perceived threat.

The visible effect of this overload management policy is that guidance updates for weapons paired with the greatest threats are issued on-time, so the most significant threats are intercepted before they can inflict damage. Guidance updates for other weapons may be delayed. The delay may result in longer flight paths and later intercept times, but will not allow hostile targets to leak through unless the system is so overloaded that even the most essential activities cannot be completed.

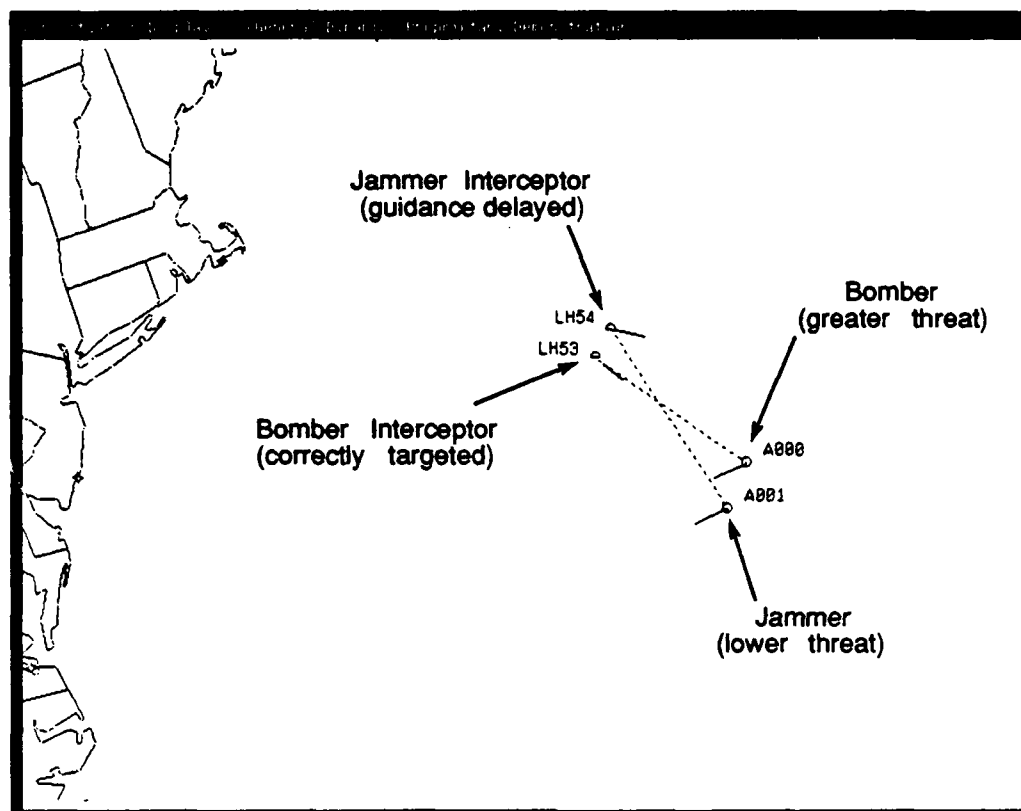


Figure 17: Overload Behavior

Figures 16 and 17 illustrate the overload handling policy. In this case, two friendly interceptors (LH53 and LH54) have been paired with a hostile bomber (A000) and a hostile jammer (A001) respectively. These pairings are indicated by dashed pairing lines between the friendly interceptors and the threats. In the normal case (Figure 16), both interceptors receive guidance updates when needed and are correctly directed toward their targets. (Track headings and speeds are indicated by the direction and length of the line emanating from the track marker.) In Figure 17, an artificial overload has been imposed on the ADSP. In the overload case, the interceptor paired with the bomber (a great threat) is given preference over the interceptor paired with the jammer (a lesser threat). As a result, the bomber is correctly targeted by LH53, while guidance updates for LH54 are delayed (LH54 is not flying toward its target and is farther away from the threat). The aggregate result of this behavior is that hostile targets tend to penetrate deeper when the system is overloaded than under normal conditions. Even under overload conditions, however, the ADSP continues to satisfy its mission.

9 Conclusions

The results of the Alpha application effort have been extremely worthwhile and very encouraging. Using Alpha, a small group of programmers was able to design and implement a realistic air defense system prototype in a period of a few months. The prototype has successfully demonstrated many of Alpha's distribution, time-driven scheduling, and survivability mechanisms, and has validated our belief that Alpha is well-suited for the design and construction of large, distributed, real-time applications. This chapter relates some of the many lessons we have learned from the effort, and describes some of the continuing work that is in progress.

9.1 Lessons Learned

9.1.1 Decentralized Computing

Alpha is designed to integrate a set of physically dispersed computer nodes into a single decentralized system. Our experience with the ADSP indicates that we have made great strides toward that goal. Objects can be dynamically created at any node, and can be invoked without knowing the node on which they actually reside. Application threads move freely across node boundaries—carrying attributes such as atomicity and time constraints with them. A major benefit of this decentralization is that it allows applications to obtain the availability and survivability benefits of physical distribution without incurring many of the difficulties often associated with conventional distributed programming.

9.1.2 Programming Model

The application designers at GD found the Alpha programming model easier to use than conventional process-based models. The object/thread paradigm allowed the design team to think in application terms when designing the structure of the software. Since there was no conversion step to translate the natural system structure into foreign constructs such as client/server processes, the overall design time was shortened. The requirements specification led quickly to an implementation that was both easy to understand and simple to modify as requirements changed.

9.1.3 Time-Driven Resource Management

Our experience with the ADSP has confirmed that C²/BM applications have timeliness requirements that are not easily expressed in terms of fixed, periodic, or hard deadlines. The coastal air defense application presented a variety of timeliness requirements. Activities such as radar plot correlation have roughly periodic arrival rates, but soft time constraints. Weapons guidance activities have dynamically changing timeliness requirements whose duration, "hardness," and importance vary as the encounter progresses. Finally, activities such as SAM launch control have strict hard deadlines, but occur aperiodically. None of these examples match the popular notion that real-time tasks are either periodic with hard deadlines or aperiodic with loose timing requirements.

In comparison, the ADSP has demonstrated that the time-driven scheduling facilities provided by Alpha are appropriate for use in large distributed systems. Time-value functions allow a variety of system timeliness requirements to be specified both simply and accurately. The best-effort scheduling algorithm used for the ADSP successfully allo-

cates processor cycles to ensure that, when possible, activities are completed on time. When not enough resources are available (e.g., because of a node failure), the overload handling capabilities of the best-effort algorithm keep the ADSP operating and allow it to continue fulfilling its primary mission requirements.

9.1.4 System Survivability

Mission-oriented systems such as the ADSP need to continue functioning even if parts of the computer system fail. Designing a system that could weather node failures was made much easier by the survivability mechanisms provided by Alpha. Alpha automatically detects node failures, returns the system to a consistent state, and reports the failure to the affected application threads via a uniform exception handling mechanism. As a result, the application designers could concentrate on developing a recovery policy that ensures that no vital information or activities are lost. Because the operating system handles many of the details, the actual implementation and testing of the recovery operations for the ADSP was relatively simple. The resulting application successfully recovers from multiple node failures and even allows repaired nodes to be re-integrated into the running system.

Our experience designing the application-specific distributed services and recovery managers has allowed us to identify additional system-level services that would aid the development of survivable applications. Future releases of Alpha will incorporate this experience in the form of system service objects that provide much of the functionality of the *DSM* and *RecoveryManager*, but in a more general and consistent manner.

9.1.5 Program Management

One unique aspect of the application effort was its timing with respect to the development of the operating system. Since Alpha was being developed in tandem with the application, good communications between the operating system group at CMU and the application team at GD were vital. The use of a dedicated liaison, who spent a significant amount of time at both sites, worked extremely well under these circumstances. The people at GD knew who to ask when questions arose (which, considering the dearth of Alpha documentation at the time, they often did). Meanwhile, the group at CMU could concentrate their efforts on developing Alpha. The liaison, being familiar with both the ADSP code and the operating system internals, could efficiently investigate bugs as they appeared, and could work to isolate and fix problems with minimal disruption of the application or operating system development.

To support the application effort, the Alpha group had to reorganize the project schedule and priorities somewhat. Implementation of some features was deferred (notably secondary storage and atomic transactions) in order to expedite the development of other features required by the ADSP. Features that were accelerated or enhanced beyond their planned level of functionality include the time-driven scheduling policies, the application programming environment, Alpha/UNIX interoperability mechanisms, and asynchronous exception handling. Any negative impact of the schedule changes, however, was far outweighed by the benefits of and experience gained from the application effort.

9.2 Future Directions

The ADSP was the first significant application developed for Alpha. One of the goals in undertaking such an early application effort was to identify areas where further research and development efforts would be most effective. As expected, the experience has highlighted several areas where research is continuing. The success of the effort has also prompted us to implement other applications in an attempt to expand our understanding of different application requirements and determine how well they are met by the mechanisms and policies provided by Alpha.

9.2.1 Related Research

Our experience with the ADSP has highlighted several areas where additional research is warranted. In particular, it seems vital that we continue our exploration of real-time scheduling techniques, object replication, and real-time, atomic transactions.

The best-effort scheduling algorithm employed in the prototype has performed well. However, Alpha researchers have identified methods of improving its ability to handle certain types of activities. In particular, two Ph.D. theses are addressing the issues of threads with dependency relationships and time-constrained threads that span multiple nodes (see [Clark 88], [Archons 88]). We have also recognized the need to develop a better methodology for deriving time-value functions and better tools for computing the execution-time estimates required by many real-time scheduling algorithms.

9.2.2 System Enhancements

The ADSP continues to be a valuable tool for experimenting with Alpha. By more thoroughly monitoring the current system and by implementing extensions and enhancements, we can learn significantly more from the air defense application. To date, only limited performance monitoring of the ADSP has been carried out. By further instrumenting the system, it should be possible to learn a great deal about the resource request patterns of a real application. This information would provide valuable insight into the nature of C²/BM applications, and would allow the Alpha designers to concentrate performance tuning efforts in areas that would make the greatest difference.

The ADSP should also provide an excellent environment for testing new Alpha functionality. The essential data repository objects would be ideal candidates for exercising such facilities as object replication and atomic transactions. Similarly, much of the functionality of the *DSM* object could gradually be migrated into more general system-level services that support system initialization, load distribution, and failure reconfiguration. Already having a large application that can benefit from these facilities will prove extremely valuable when testing their functionality.

Because the ADSP was designed as a prototype, it would be relatively simple to enhance the system by increasing the fidelity of many of its functions. Many of the mission planning objects could be changed to use more sophisticated algorithms or could be converted into decision-support systems for a human operator. The system could also be enhanced to support additional functions such as multi-sensor fusion. These types of enhancements should especially interest application developers who could gain experience designing command and control systems in the Alpha context.

9.2.3 Future Applications

Experience with the ADSP indicates that the best-effort scheduler works well in large real-time systems. However, the complexity of the ADSP limits the direct visibility of scheduling decisions, and makes it difficult to quantify the performance of the scheduler. For these reasons, the Alpha designers are developing an application designed explicitly to test the behavior and performance of the Alpha scheduling facility. The new application will allow us to monitor the decisions made by the scheduler in a more controlled environment, and will allow us to compare the results obtained using alternate scheduling algorithms. The initial application uses Alpha to move a set of simulated paddles that reflect balls travelling through the air. Different scheduling algorithms move the paddles in different patterns—allowing a direct, graphical display of the processor allocation policies. A separate technical report will detail these experiments when they have been completed.

In addition to specialized application systems such as the scheduling application, we anticipate that other real-time, C²/BM prototypes will be built using later generations of Alpha. Planning for such application systems, involving several industrial partners, is already in progress at Concurrent Computer Corporation.

References

- [Archons 88] Archons Project
Alpha Preview: A Briefing and Technology Demonstration for DoD.
Presentation Notes, Archons Project Technical Report #88031, Department of Computer Science, Carnegie Mellon University, March, 1988.
- [Clark 88] Clark, R. K.
Scheduling Dependent Real-Time Activities.
Ph.D. Thesis Proposal, Department of Computer Science, Carnegie Mellon University, October, 1988.
- [Cox 86] Cox, B. J.
Object-Oriented Programming.
Addison-Wesley, Reading, Massachusetts, 1986.
- [Fairley 85] Fairley, R. E.
Software Engineering Concepts.
McGraw-Hill Series in Software Engineering and Technology, McGraw-Hill, New York, 1985.
- [Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June, 1975.
- [Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, May, 1986.
- [Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.
- [Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie Mellon University, January, 1988.
- [Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie Mellon University, February, 1988.
- [Northcutt 88c] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie Mellon University, March, 1988.
-

References

- [Archons 88] Archons Project
Alpha Preview: A Briefing and Technology Demonstration for DoD.
Presentation Notes, Archons Project Technical Report #88031, Department of Computer Science, Carnegie Mellon University, March, 1988.
- [Clark 88] Clark, R. K.
Scheduling Dependent Real-Time Activities.
Ph.D. Thesis Proposal, Department of Computer Science, Carnegie Mellon University, October, 1988.
- [Cox 86] Cox, B. J.
Object-Oriented Programming.
Addison-Wesley, Reading, Massachusetts, 1986.
- [Fairley 85] Fairley, R. E.
Software Engineering Concepts.
McGraw-Hill Series in Software Engineering and Technology, McGraw-Hill, New York, 1985.
- [Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June, 1975.
- [Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, May, 1986.
- [Northcutt 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.
- [Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie Mellon University, January, 1988.
- [Northcutt 88b] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Programming Model.
Archons Project Technical Report #88021, Department of Computer Science, Carnegie Mellon University, February, 1988.
- [Northcutt 88c] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie Mellon University, March, 1988.

-
- [Northcutt 88d] Northcutt, J. D. and Clark, R. K.
The Alpha Operating System: Kernel Internals.
Archons Project Technical Report #88051, Department of Computer
Science, Carnegie Mellon University, May, 1988.
- [Northcutt 88e] Northcutt, J. D.
The Alpha Operating System: Programming Utilities.
Archons Project Technical Report #88041, Department of Computer
Science, Carnegie Mellon University, April, 1988.
- [Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer
Science, Carnegie Mellon University, April, 1988.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK.

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTION LIST

addresses	number of copies
RADC/COTD ATTN: Thomas F. Lawrence Griffiss AFB NY 13441-5700	10
Carnegie-Mellon University Dept of Computer Science Dept of Electrical Engineering Pittsburgh PA 15213-3890	5
RADC/DOVL Technical Library Griffiss AFB NY 13441-5700	1
Administrator Defense Technical Info Center DTIC-FDA Cameron Station Building 5 Alexandria VA 22304-6145	2
Strategic Defense Initiative Office Office of the Secretary of Defense Wash DC 20301-7100	2
RADC/COTD Building 3, Room 16 Griffiss AFB NY 13441-5700	1
AFCSA/SAMI ATTN: Miss Griffin 10363 Pentagon Washington DC 20330-5425	1
HQ USAF/SCIT Pentagon Washington DC 20330-5190	1

SAF/AQSC
Pentagon 4D-267
Washington DC 20330-1000

1

Fleet Analysis Center
GIDEP Operations Center
ATTN: Mr. E. Richards
Code 30G1
Corona CA 91720

1

HQ AFSC/XTS
Andrews AFB MD 20334-5000

1

HQ SAC/SCPT
OFFUTT AFB NE 68113-5001

1

DTESA/RQE
ATTN: Mr. Larry G. McManus
Kirtland AFB NM 87117-5000

1

HQ TAC/DRIY 1
ATTN: Mr. Westerman
Langley AFB VA 23665-5001

HQ TAC/DOA 1
Langley AFB VA 23665-5001

HQ TAC/DRCA 1
Langley AFB VA 23665-5001

ASD/ENEMS 2
Wright-Patterson AFB OH 45433-6503

SM-ALC/MACEA 1
ATTN: Danny McClure
BLDG 250N
McClellan AFB CA 95652

ASD/AFALC/AXAE 1
ATTN: W. H. Dungey
Wright-Patterson AFB OH 45433-6533

WRDC/AAAI 1
Wright-Patterson AFB OH 45433-6533

AFIT/LDEE 1
Building 640, Area B
Wright-Patterson AFB OH 45433-6583

WRDC/MLPO 1
Wright-Patterson AFB OH 45433-6533

WRDC/MLTE 1
Wright-Patterson AFB OH 45433

WRDC/FIES/SURVIAC 1
Wright-Patterson AFB OH 45433

AAMRL/HE 1
Wright-Patterson AFB OH 45433-6573

Air Force Human Resources Lab 1
Technical Documents Center
AFHRL/LRS-TDC
Wright-Patterson AFB OH 45433

AFHRL/OTS 1
Williams AFB AZ 85240-6457

AUL/LSE 1
Maxwell AFB AL 36112-5564

HQ Air Force SPACECOM/XPYS 1
ATTN: Dr. William R. Matoush
Peterson AFB CO 80914-5001

Defense Communications Engr Center 1
Technical Library
1360 Wiehle Avenue
Reston VA 22090-5500

C3 Division Development Center Marine Corps Development & Education Command Code DIOA Quantico VA 22134-5080	2
AFLMC/LGY AF Logistics Management Center Chief, System Engineering Division Gunter AFS AL 36114	1
US Army Strategic Defense Command DASD-H-MPL PO Box 1500 Huntsville AL 35807-3801	1
Commanding Officer Naval Avionics Center Library D/765 Indianapolis IN 46219-2139	1
Commanding Officer Naval Training Systems Center Technical Information Center Building 2068 Orlando FL 32813-7100	1
Commanding Officer Naval Ocean Systems Center Technical Library Code 46423 San Diego CA 92152-5000	1
Commanding Officer Naval Weapons Center Technical Library Code 3433 China Lake CA 93555-6001	1
Superintendent Naval Post Graduate School Code 1424 Monterey CA 93943-5000	1
Commanding Officer Naval Research Laboratory Code 2627 Washington DC 20375-5000	2

Service & Naval Warfare Systems COMM 1
PMW 153-3DP
ATTN: R. Savarese
Washington DC 20363-5100

Commanding Officer 2
US Army Missile Command
Redstone Scientific Info Center
AMSMI-RD-CS-R (Documents)
Redstone Arsenal AL 35898-5241

Advisory Group on Electron Devices 2
Technical Info Coordinator
ATTN: Mr. John Hammond
201 Varick Street - Suite 1140
New York NY 10014

Los Alamos Scientific Laboratory 1
Report Librarian
ATTN: Mr. Dan Baca
PO Box 1563, MS-P364
Los Alamos NM 87545

Rand Corporation 1
Technical Library
ATTN: Ms. Doris Helfer
PO Box 2133
Santa Monica CA 90406-2138

AEDC Library 1
Technical Reports File
MS-100
Arnold AFS TN 37389-9998

USAG 1
ASH-PCA-CRT
Ft. Huachuca AZ 85613-6000

1339 EIG/EIET 1
ATTN: Mr. Kenneth W. Irby
Keesler AFB MS 39534-6348

JTFPO-TD 1
Director of Advanced Technology
ATTN: Dr. Raymond F. Freeman
1500 Planning Research Drive
McLean VA 22102

HQ ESC/CWPP
San Antonio TX 78243-5000

1

AFEWC/ESRI
San Antonio TX 78243-5000

3

435 EIG/EIR
ATTN: M Craft
Griffiss AFB NY 13441-6348

1

ESD/XTP
Hanscom AFB MA 01731-5000

1

ESD/AVSE
Building 1704
Hanscom AFB MA 01731-5000

2

HQ ESD SYS-2
Hanscom AFB MA 01731-5000

1

Software Engineering Institute
Joint Program Office
ATTN: Major Dan Burton, USAF
Carnegie Mellon University
Pittsburgh PA 15213-3390

1

Director
NSA/CSS
T513/TDL
ATTN: Mr. David Marjarum
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
W166
Fort George G. Meade MD 20755-6000

1

1

Director
NSA/CSS
DEFSMAC
ATTN: Mr. James E. Hillman
Fort George G. Meade MD 20755-6000

Director
NSA/CSS
R5
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
R8
Fort George G. Meade MD 20755-6000

1

Director
NSA/CSS
S21
Fort George G. Meade MD 20755-6000

1

Director NSA/CSS R523 Fort George G. Meade MD 20755-6000	2
DOD Computer Security Center C4/TIC 9800 Savage Road Fort George G. Meade MD 20755-6000	1
Honeywell - SSDC ATTN: Mr. Jeremy Norton 10009 Boone Ave MS: Mn63-C040 Golden Valley MN 55427	1
SDI/S-P1-BM ATTN: Cmdr Korajo The Pentagon Wash DC 20301-7100	1
SDIO/S-PL-BM ATTN: Capt Johnson The Pentagon Wash DC 20301-7000	1
SDIO/S-PL-BM ATTN: Lt Col Rindt The Pentagon Wash DC 20301-7100	1
IDA (SDIO Library) ATTN: Mr. Albert Perrella 1801 N. Beauregard Street Alexandria VA 22311	1
SAF/AQSD ATTN: Maj M. K. Jones The Pentagon Wash DC 20330	1
AFSC/CV-D ATTN: Lt Col Flynn Andrews AFB MD 20334-5000	1

HQ SD/XR ATTN: Col Heimach PO Box 92960 Worldway Postal Center Los Angeles CA 90009-2960	1
HQ SSD/CNC ATTN: Col O'Brien PO Box 92960 Worldway Postal Center Los Angeles CA 90009-2960	1
HQ SD/CNCI ATTN: Col Collins PO Box 92960 Worldway Postal Center Los Angeles CA 90009-2960	1
HQ SD/CNCIS ATTN: Lt Col Pennell PO Box 92960 Worldway Postal Center Los Angeles CA 90009-2960	1
ESD/AT ATTN: Col Ryan Hanscom AFB MA 01731-5000	1
ESD/ATS ATTN: Lt Col Oldenberg Hanscom AFB MA 01731-5000	1
ESD/ATN ATTN: Col Leib Hanscom AFB MA 01731-5000	1
AFSTC/XPX (Lt Col Detucci) Kirtland AFB NM 87117	1
USA SDC/DASD-H-SB (Larry Tubbs) P. O. Box 1500 Huntsville AL 35807	1

AFSPACECOM/XPD 1
ATTN: Maj Roger Hunter
Peterson AFB CO 80914

GE SDI-SEI 1
ATTN: Mr. Ron Marking
1787 Century Park West
Bluebell PA 194422

MITRE Corp 1
ATTN: Dr. Donna Cuomo
Bedford MAS 01730

SSD/CNI 1
ATTN: Lt Col Joe Rouge
P. O. Box 92960
Los Angeles AFB CA 90009-2960

NTB JPO 1
ATTN: Maj Don Ravenscroft
Falcon AFB CO 80912

Ford Aerospace Corp 1
c/o Rockwell International
ATTN: Dr. Joan Schulz
1250 Academy Park Loop
Colorado Springs CO 80910

Essex Corp 1
ATTN: Dr. Bob Mackie
Human Factors Research Div
5775 Dawson Ave
Goleta CA 93117

Naval Air Development Ctr 1
ATTN: Dr. Mort Metersky
Code 300
Warminster PA 189974

RJO Enterorises 1
ATTN: Mr. Dave Israel
1225 Jefferson Davis HWY
Suite 300
Arlington VA 22202

GE SDI SEI
ATTN: Mr. Bill Bensch
1707 Century Park West
Bluebell PA 19422

HQ AFOTEC/OAHS
ATTN: Dr. Samuel Charlton
Kirtland AFB NM 87117

ESD/XTS
ATTN: Lt Col Joseph Toole
Hanscom AFB MA 01731

SDIO/ENA
ATTN: Col R. Worrell
Pentagon
Wash DC 20301

USA-SDC CSSD-H-SBE
ATTN: Mr. Doyle Thomas
Huntsville AL 35807

HQ AFSPACECOM/DOXP
ATTN: Capt Mark Terrace
Stop 7
Peterson AFB CO 80914

BRN Systems & Technology
ATTN: Dr. Dick Pew
70 Fawcett St
Cambridge MA 02138

ESD/XTI
ATTN: Lt Col Paul Monico
Hanscom AFB MA 01730

CSSD-H-SB
ATTN: Mr. Larry Tubbs
Commander USA SDC
PO Box 1500
Huntsville AL 35807

1

1

1

1

1

1

1

1

1

USSPACECOM/J5B
ATTN: Lt Col Harold Stanley
Peterson AFB CO 80914

1

NTB JPO
ATTN: Mr. Nat Sojourner
Falcon AFB CO 80912

1

RADC/COT
ATTN: Mr. Ronald S. Raposo
Griffiss AFB NY 13441

1

Bonnie McDaniel, MDE
313 Franklin St
Huntsville AL 35801

1

The Aerospace Corporation
ATTN: Mr. George Gilley
ML-046
PO Box 92957
Los Angeles CA 90530

5

AF Space Command/XPXIS
Peterson AFB CO 80914-5001

1

AFJTEC/XPP
ATTN: Capt Wrobel
Kirtland AFB NM 87117

1

Director NSA (V43)
ATTN: George Hoover
9300 Savage Road
Ft George G. Meade MD 20755-6000

1

SSD/CNIR
ATTN: Capt Brandenburg
PO BOX 92960-2960
LOS ANGELES CA 90009-2960

1

Advanced System Technologies 1
ATTN: Duane R. Ball
5113 Leesburg Pike, Suite 514
Falls Church VA 22041

Odyssey Research Associates, Inc. 1
ATTN: Doug Weber
301A Harris B. Dates Dr
Ithaca NY 14850-1313

SRI International 1
ATTN: Teresa Lunt, BN169
333 Ravenswood Ave
Menlo Park CA 94025

SRI International 1
ATTN: Mathew Morgenstern, BN 162
333 Ravenswood Ave
Menlo Park CA 94025

George Mason University 1
ATTN: Prof Sushil Jajodia
ISSE Department
4400 University Drive
Fairfax VA 22030-4444

GE (Advanced Technology Labs) 1
ATTN: L. D. Alexander
Bldg 145-2, Route 38
Moorestown NJ 08057

Concurrent Computer Corp 1
ATTN: E. Douglas Jensen
1 Technology Way
Westford MA 01886

Xerox Advanced Info Technology 1
ATTN: Barbara Blaustein
7900 West Park Dr, Suite 400
McLean VA 22102

Dove Electronics, Inc. 1
ATTN: John Dove
227 Liberty Plaza
Rome NY 13440

W. W. Chu Associates 1
ATTN: Dr. Wesley Chu
16794 Charmel Lane
Pacific Palisades CA 90272

U. S. Army CECOM 1
ATTN: Lakshmi V. Rebbapragada
Center for C3 Systems
AMSEL-RD-C3-IR
Ft Monmouth NJ 07703

M/A 473 1
NASA-Langley Research Center
ATTN: Nicholas D. Murray
Hampton VA 23665

Trusted Information Systems, Inc. 1
ATTN: Steve Walker
3060 Washington Rd
Glenwood MD 21738

Advanced Decision Systems 1
ATTN: Andrew Cromerty
1500 Plymouth St
Mountain View CA 94043

Genini Computers Inc. 1
ATTN: Dr. Roger Schell
ICS Division
PO Box 222417
Carmel CA 93922

Naval Ocean Systems Center 1
ATTN: Les Anderson
271 Catalina Blvd, Code 413
San Diego CA 92151

Carnegie Mellon University 1
Dept of Computer Science
ATTN: Ray Clark
Schenley-Wean Hall
Pittsburgh PA 15213-3890

Univ of Maryland 1
Dept of Computer Science
ATTN: Ashok K. Agrowala
College Park MD 20742

The Nitre Corporation
ATTN: Myra Jean Prella
Burlington Rd
Bedford MA 01730

1



MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.